Self-Stabilizing Algorithms for Constructing Distributed Spanning Structures

Yvan Rivierre

UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Arrêté ministériel : 7 août 2006

Présentée par

Yvan RIVIERRE

Thèse dirigée par **Florence MARANINCHI**
et co-encadrée par **Fabienne CARRIER**
et **Stéphane DEVISMES**

préparée au sein du laboratoire **VERIMAG**
et de l'**École Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique**

Algorithmes auto-stabilisants pour la construction de structures couvrantes réparties

Self-Stabilizing Algorithms for Constructing Distributed Spanning Structures

Thèse soutenue publiquement le **12 décembre 2013**,
devant le jury composé de :

Alain COURNIER
Université de Picardie Jules Verne, Président
Ajoy Kumar DATTA
Pr, University of Nevada, Las Vegas, Rapporteur
Franck PETIT
Pr, Université Pierre et Marie Curie, Rapporteur
Fabien MATHIEU
HDR, Alcatel-Lucent Bell Labs France, Examinateur
Florence MARANINCHI
Pr, Institut Polytechnique de Grenoble, Directrice de thèse
Fabienne CARRIER
McF, Université Joseph Fourier, Co-encadrante de thèse
Stéphane DEVISMES
McF, Université Joseph Fourier, Co-encadrant de thèse

À la mémoire de Daniel, Zandra et Olivier.

Remerciements

Je souhaite adresser ici mes plus vifs remerciements à l'ensemble des personnes qui m'ont aidé de près ou de loin dans mes études, dont la présente thèse de doctorat constitue un aboutissement. En raison de votre très grand nombre, je vous prie par avance de bien vouloir m'excuser de mes éventuels oublis.

Merci à Stéphane Devismes d'avoir partagé son enthousiasme et sa curiosité pour l'algorithmique répartie, ce qui m'a donné à la fois l'envie et l'opportunité de m'investir dans ce domaine de recherche en particulier. Merci à Fabienne Carrier d'avoir partagé son expérience de recherche et son sens de la pédagogie, ce qui m'a grandement encouragé et aidé à rendre plus compréhensibles mes travaux trop souvent obscurs pour les néophytes. Votre duo d'encadrement de choc a su me guider et me soutenir avec une patience infinie tout au long de mon doctorat. Merci aussi à Florence Maraninchi de nous avoir fait confiance en acceptant de diriger cette thèse et d'avoir été attentive dans les moments clés.

Merchi grammint à Ajoy Kumar Datta et Franck Petit d'avoir accepté d'être les rapporteurs de ma thèse. Cela a aussi été un plaisir de travailler avec vous à de multiples occasions, avant et pendant mon doctorat, parfois au péril de votre vie.

Merci également aux deux autres membres du jury : Alain Cournier de m'avoir fait l'honneur de présider ce jury et Fabien Mathieu d'y avoir apporté un regard extra-universitaire.

Merci bien sûr aux autres personnes qui ont collaboré à mes travaux de thèse, Karel Heuretefeux et Lawrence L. Larmore. Vos contributions complémentaires ont eu une importance majeure pour les résultats présentés dans cette thèse.

Merci par ailleurs à l'ensemble des contributrices et contributeurs à TEX, LATEX, PGF/TikZ, gnuplot, Xfig, Vim, GraphViz, Sinalgo, WSNet, Linux et au projet GNU, pour ces logiciels libres qui ont facilité la réalisation de mon travail de recherche.

Merci aux collègues de recherche et d'enseignement dont le sens de l'écoute, les conseils simples ou les échanges constructifs m'ont particulièrement aidé à mener à bien mon doctorat, je pense notamment à Laurent Mounier, Pascal Lafourcade, Simplice Djoko-Djoko, Laurent Fousse, Antoine Gerbaud et Sanjay Rawat.

Merci à Pierre-Louis Aublin, Nicolas Berthier, Mathilde Duclos, Jannik Dreier, Yean-Li Ho et tous mes pairs de promotion pour leur esprit de camaraderie.

Teşekürler aux enseignants et chercheurs qui m'ont accompagné en fin de deuxième cycle à l'Université de Galatasaray, Jean-Claude Fernandez, Chantal Cherifi, Hocine Cherifi, Vincent Labatut et Günce Orman.

Merci également aux personnes qui m'ont aidé à mettre le pied à l'étrier ou à remonter en selle pendant mon parcours universitaire, je pense notamment à Romain Archambault, Josselyne Cheraton, Karol Proch, Alain Mirgaux, Philippe Bizard et Christian Boitet.

Enfin, j'adresse un immense merci à mes ami-e-s qui se reconnaîtront, à toute ma famille et plus particulièrement à mes frères et à mes parents.

Résumé substantiel

Cette thèse est essentiellement rédigée en anglais, de façon à être lisible par l'ensemble de mes rapporteurs et de mes collaborateurs. Ce résumé en français tente de la rendre plus accessible au lectorat profane dans le domaine de l'auto-stabilisation et francophone. Il requiert au minimum des notions en théorie des graphes et en théorie de la complexité algorithmique. Le plan du résumé correspond à celui de la thèse, partie par partie, afin de faciliter le passage de l'un à l'autre pour approfondissement.

Introduction

Depuis qu'ils existent, les systèmes informatiques ont toujours été la proie de fautes provenant de leur environnement, les induisant en erreur, pouvant ainsi aller jusqu'à perturber leur fonctionnement voire même provoquer des pannes dans le service qu'ils fournissent. Un système ne doit donc pas seulement être correct dans son fonctionnement interne, il doit aussi être conçu pour tolérer d'éventuelles fautes dont la cause peut être externe.

À cet effet, la *tolérance aux fautes* peut être envisagée de deux façons radicalement différentes. D'abord, un système peut être conçu, quand cela est possible, pour empêcher toute faute de l'affecter et de causer des erreurs. Cette approche est dite « robuste » et entraîne généralement soit des coûts supplémentaires non négligeables en espace mémoire et/ou en temps de calcul, soit des fonctionnalités amoindries. Autrement, un système peut-être conçu pour se remettre des fautes en corrigeant de lui-même les erreurs qu'elles ont causées. Le coût est généralement beaucoup moins élevé que pour l'approche précédente. La contrepartie est que le service normalement assuré par le système ne peut être garanti entre le moment où les fautes provoquent des erreurs et celui où le système récupère complètement.

C'est dans cette dernière approche que s'inscrit l'*auto-stabilisation* qui a été proposée pour la première fois par Dijkstra [Dij74]. Elle suppose que les fautes sont transitoires, c'est-à-dire, que les fautes n'affectent le système que dans de rares et relativement courtes périodes de temps. Cependant, elle ne fait aucune hypothèse sur la sévérité et le nombre de fautes. En fait, l'auto-stabilisation garantit que le système puisse récupérer depuis n'importe quel état. Ce paradigme a été développé dans le contexte des *systèmes répartis*.

Un système est dit réparti quand il est composé de plusieurs entités. Chaque entité exécute son propre programme indépendamment (sans dépendre d'un contrôle central) et peut communiquer avec un sous-ensemble arbitraire d'autres entités (selon la topologie du système réparti). Malgré cela, un système réparti apparaît dans son ensemble comme un seul système au regard de ses utilisateurs. Le rassemblement de tous les programmes locaux est appelé un *algorithme réparti* ; c'est la recette qui permet à un système réparti de fournir le service voulu à l'utilisateur.

De façon à surmonter le chaos apparent qui prévaut dans un système réparti, les applications ont généralement besoin d'une sorte de coordination entre les entités. Celle-ci peut être obtenue par l'intermédiaire de *structures couvrantes réparties*, c'est-à-dire de structures de données qui répartissent les informations collectées à travers l'ensemble du système. Dans la littérature de l'auto-stabilisation, les plus connues de ces structures sont certainement l'arbre couvrant et le partitionnement. La principale raison d'être de la première structure est d'éviter les boucles infinies de communication entre les entités. La seconde organise hiérarchiquement les entités, permettant l'implémentation d'applications réparties de plus haut niveau telles que le routage, le calcul réparti ou encore les bases de données réparties... Ce type de structure est particulièrement utile dans les réseaux à grande échelle. Plus les entités sont nombreuses dans un système réparti, plus il est probable qu'au moins une de ces entités soit affectée par des fautes. Par exemple, Internet compte des milliards d'appareils, dont une bonne part de machines bon marché particulièrement sujettes aux défauts ; il est donc certain que ce système est la proie régulière de fautes.

Cette thèse porte plus particulièrement sur l'auto-stabilisation d'algorithmes répartis construisant des structures couvrantes réparties d'un système réparti. Après avoir motivé son étude et posé ses prérequis théoriques (Partie I, page viii), elle abordera deux problèmes auxiliaires (Partie II, page xx). Enfin, elle traitera principalement les trois problèmes suivants :

- la construction d'une k-partition (Partie III, page xxii) qui organise hiérarchiquement les entités d'un système réparti,
- la construction d'une (f, g)-alliance (Partie IV, page xxvii) qui généralise plusieurs types d'alliance dont l'ensemble dominant,
- la construction d'un index réparti (Partie V, page xxxi) qui attribue un rang à chaque entité du système selon un ensemble réparti de valeurs en entrée.

Pour chacun de ces problèmes, nous prêtons attention à la fois à la correction et à l'efficacité des solutions que nous proposons. Dans le cas du k-partitionnement, nous évaluons aussi chacune de nos différentes solutions à l'aide de simulations.

Partie I : Contexte

Cette première partie est consacrée à présenter les motivations de cette thèse ainsi que les outils théoriques employés ici et le modèle de calcul avec lequel sont développés les algorithmes de cette thèse.

Brève genèse des systèmes répartis

Bien avant l'avènement de l'électronique, tous les calculs étaient faits manuellement, parfois avec l'aide d'un boulier ou d'une règle. Les calculs les plus complexes pouvaient être délégués à quelqu'un d'autre dont c'était le métier. Ainsi donc, les premières « calculatrices » étaient humaines et pratiquaient déjà le calcul réparti.

En 1946, le premier calculateur électronique généraliste fut créé et appelé ENIAC [Bur02]. Ce fut le premier ordinateur conçu comme une machine de Turing universelle, c'est-à-dire, capable de simuler (avec des ressources suffisantes) n'importe quelle autre machine. Rapidement ensuite, vint le besoin de faire communiquer les ordinateurs entre eux. Dans les années 1960, les réseaux d'ordinateurs marquent un tournant avec l'introduction des paquets alternés [Bar64], réduisant de manière drastique le coût des communications de données électroniques. À la même période, la puissance de calcul des ordinateurs est améliorée par l'emploi de multiprocesseurs et par la programmation multitâche [Cri63], réduisant ainsi le coût des ordinateurs. Les années 1980 virent l'émergence d'Internet, le plus célèbre réseau d'ordinateurs sur Terre, qui connecte aujourd'hui des milliards d'appareils électroniques entre eux. Mais il existe aussi des réseaux plus communs ou à venir tels que les réseaux de capteurs sans fil, les réseaux construits sur une architecture pair à pair, les grilles de calcul, les réseaux mobiles *ad hoc*.

Le système réparti comme abstraction

Un *système réparti* est une collection interconnectée d'entités indépendantes, qui apparaît comme un seul système à ses utilisateurs. Ainsi, c'est un concept abstrait qui correspond à de nombreux cas réels de systèmes répartis. Chaque terme de cette définition est expliqué dans les paragraphes suivants.

D'abord, une *entité* représente ici une unité de calcul d'un type quelconque tel que : un ordinateur, le cœur d'un microprocesseur, un processus dans le contexte d'un système d'exploitation multitâche... L'*indépendance* des entités est triple : elles sont asynchrones, autonomes et sans connaissance globale. En effet, les entités peuvent ne pas être équipées d'une horloge synchronisée et chaque entité peut être influencée par son propre environnement. Ainsi, chaque entité peut effectuer des calculs et des communications à sa propre vitesse, c'est-à-dire que les entités sont *asynchrones*. Les entités sont dites *autonomes* au sens où chacune d'entre elles embarque son propre programme et ne dépend d'aucun système de contrôle central. Le programme local de chaque entité ne contient habituellement pas d'information globale de quelque genre que ce soit et plus particulièrement concernant le système réparti lui-même. Les paramètres globaux, tels que le nombre d'entités dans le système, sont susceptibles de changer d'un déploiement à l'autre et à travers le temps. Les entités sont dites *sans connaissance globale*, car chaque entité exécute son propre programme d'après sa seule connaissance locale.

De plus, ces entités peuvent communiquer entre elles, *connectées* par divers moyens tels que les paires de fils torsadées, les ondes radio de haute fréquence... Cela ne signifie pas que chaque entité est directement connectée à n'importe quelle autre. Pourtant chaque entité est capable de diffuser indirectement une information à l'ensemble du système réparti si celle-ci est transmise de proche en proche.

Enfin, les utilisateurs de systèmes répartis n'ont pas besoin de connaître leur architecture interne pour bénéficier de leur service. Cela permet une bonne modularité dans l'implémentation du service. Par exemple, nous ne nous préoccupons pas de

savoir si les services postaux rassemblent tout le courrier dans un seul dépôt central ou s'il y a des dépôts décentralisés pour prendre en charge les lettres que nous envoyons et recevons couramment. De même, la plupart des internautes ne se préoccupent pas de comment la page Web qu'ils ont demandé est construite et leur est délivrée. Ce sont deux exemples parmi tant d'autres, avec ou sans ordinateurs. L'utilisation des systèmes répartis est donc monnaie courante, et ce depuis très longtemps.

Problèmes répartis fondamentaux

Algorithmes parallèles et algorithmes répartis. Un *algorithme réparti* est conçu pour s'exécuter sur un système réparti donné pour réaliser une tâche globale à ce système, en utilisant des entrées qui sont dispersées parmi ses entités. Ici, nous distinguons les algorithmes parallèles des algorithmes répartis. Le parallélisme est une technique pour effectuer des très grands calculs en les découpant en calculs plus petits qui peuvent être effectués indépendamment sur diverses entités. Cela permet d'obtenir un résultat au calcul global beaucoup plus rapidement. Cependant, les algorithmes parallèles sont généralement exécutés dans un environnement contrôlé, c'est-à-dire que le calcul est décentralisé mais son contrôle reste central. Les algorithmes répartis vont au-delà de la mutualisation des puissances de calcul des entités réalisée par les algorithme parallèles dans un environnement sous contrôle. Ils privilégient l'indépendance entre les entités, en particulier, les entités sont toujours supposées être autonomes. Cela permet plus de flexibilité pour leur implémentation et un déploiement plus facile. Cependant, ils doivent résoudre d'autres types de problèmes.

Problèmes classiques dans les systèmes répartis. Certains de ces problèmes ont été identifiés comme fondamentaux pour construire des applications de plus haut niveau dans les systèmes répartis. Nous en donnons ci-dessous plusieurs exemples qui illustrent les principales caractéristiques des systèmes répartis.

Routage. Il est souvent nécessaire d'envoyer des informations d'une entité d'un système réparti vers toutes les autres entités de ce système. Cependant, il faut rappeler que toutes les paires d'entités d'un système réparti ne sont pas forcément directement reliées. Pour contourner ce problème, elles peuvent communiquer de manière indirecte, par le biais d'une chaîne d'entités transmettant l'information entre elles. Habituellement, chaque entité possède une table, dite de routage, permettant de répondre à la forme suivante : pour chaque destination finale, par quelle entité voisine dois-je faire passer l'information ? Le problème du *routage* est de répondre à cette question pour chaque entité du système et pour chaque destination possible. Cela devient encore plus difficile dans le cas d'un réseau dynamique, c'est-à-dire quand certaines entités ou liens entre ces entités peuvent être ajoutés ou supprimés dans le temps, comme c'est le cas pour Internet. Rajaraman a donné une bonne vue d'ensemble sur ce sujet [Raj02]. Le routage est une application majeure de nos contributions au k-partitionnement qui sont présentées dans la partie III.

Allocation de ressources. Lorsqu'une application a besoin d'un certain nombre de ressources pour son propre fonctionnement, la question suivante émerge : Comment être sûr que toutes ces ressources seront à disposition de cette application en même temps ? Cela relève à la fois d'un problème de concurrence et de résolution de conflits. Les ressources sont habituellement attachées à un système réparti en quantité restreinte. En fait, elles ne sont généralement utilisables que par certaines entités de ce système. Le nombre d'entités est très grand comparé au nombre de ressources. Une ressource peut être utilisée au plus par une entité à la fois. Pour couronner le tout, elles peuvent même changer d'entité d'attachement au système réparti. À cause du manque de connaissance globale, une entité qui a besoin d'utiliser des ressources doit effectuer une recherche à travers le système réparti. Typiquement, les ressources peuvent être des périphériques. Un cas particulier mais intéressant d'allocation de ressources est le *problème du dîner des philosophes* brièvement introduit par Dijkstra [Dij65] et approfondi par Hoare [Hoa85]. Un cas plus général du problème d'allocation de ressources est proposé par Chandy et Misra [CM84].

Exclusion mutuelle. Voici un problème bien connu d'allocation de ressources. Il n'y a ici qu'une seule ressource et elle ne peut être utilisée au plus que par une seule entité à la fois. De façon à remplir cette condition, les entités doivent collaborer entre elles, de façon à savoir quand elles peuvent utiliser la ressource et quand elles ne doivent pas l'utiliser. Même dans ce cas plutôt simplifié, le problème n'est pas trivial, car les entités peuvent être asynchrones et n'avoir qu'une connaissance locale du système réparti. Lamport est l'auteur d'un des articles fondateurs en la matière [Lam78].

Consensus. Un système réparti peut parfois avoir besoin que toutes ses entités détiennent une même valeur à l'usage d'une application ultérieure. Comme chaque entité peut indépendamment proposer une valeur différente, elles doivent s'accorder ensemble sur une seule valeur. Dans le *problème du consensus*, chaque entité se voit initialement donner une valeur booléenne et toutes les entités doivent se mettre d'accord sur la même valeur booléenne en respectant les contraintes suivantes : la valeur finale doit être choisie parmi les valeurs initiales ; chaque entité peut choisir une valeur au plus une fois ; et enfin, chaque entité doit finir par choisir une valeur. Bien que l'énoncé de ce problème semble plutôt simple, il est en fait impossible à résoudre dans de nombreux cas, comme l'ont montré Fischer, Lynch, Merritt et Paterson [FLM85, FLP85].

Élection de leader. Rappelons nous que dans un système réparti, sans autre hypothèse, il n'y a pas de contrôle central. Pour de nombreuses applications, il est très pratique qu'une seule entité prenne les décisions pour toutes les autres. L'*élection de leader* est le problème de distinguer une entité unique parmi l'ensemble des entités d'un système réparti. Lelann a introduit ce problème pour la première fois [LL77, page 158]. Élire un leader parmi les entités d'un système réparti permet qu'une entité puisse prendre les décisions pour

l'ensemble du système, c'est-à-dire, avoir un contrôle centralisé du système réparti. Il est à noter que c'est également un bon moyen pour obtenir la synchronisation entre toutes les entités.

Structures réparties

Une *structure répartie* est une façon d'organiser un système réparti. Ce type d'organisation est fréquemment utilisé par des applications de plus haut niveau. Cela devient particulièrement intéressant dans le cas des réseaux à grande échelle. Cependant, elles sont également terriblement difficiles à construire, car elles sont globales au système réparti par nature, alors que les entités n'ont qu'une connaissance locale du système. Voici trois exemples de structures réparties, pour lesquelles un algorithme réparti de construction est proposé dans cette thèse.

Arbre couvrant. Comme vu dans le problème de routage précédent, il est souvent utile que toutes les unités puissent communiquer entre elles, même si cela doit se faire de manière indirecte, par exemple au travers d'une chaîne d'entité qui les relie. Évidemment, cette chaîne doit être de longueur finie. Mais ce n'est pas toujours le cas, si la chaîne contient des cycles, cela peut entraîner la communication à passer une infinité de fois par la même entité. Par exemple, ceci peut se produire si il y a au moins trois entités distinctes entre les deux entités qui doivent communiquer ensemble.

De façon informelle, un *arbre* est un sous-réseau d'un système réparti, tel que pour chaque paire d'entités, il y ait une et une seule chaîne d'entités distinctes entre elles, c'est-à-dire, sans cycle. Quand toutes les entités d'un système réparti font partie d'un arbre, on dit que c'est un arbre *couvrant* du système réparti.

Dans cette thèse, nous considérons toujours qu'un arbre est enraciné, c'est-à-dire, qu'une de ses entités est distinguée. Cette entité unique pour un arbre couvrant est appelée la *racine*. En raison de son caractère unique, la racine est aussi le leader du système réparti tel que défini dans le problème d'élection de leader mentionné ci-avant. Il n'est pas surprenant que les deux problèmes soient étroitement liés. En fait, la plupart des solutions au problème d'élection de leader sont proposées conjointement avec une construction d'arbre couvrant. Une vue d'ensemble de ce problème, dans le contexte de l'auto-stabilisation, est disponible [Gär03]. Nous étudions une instance spécifique d'arbre couvrant dans la partie II.

Partitionnement. Une autre façon d'obtenir une organisation hiérarchique d'un système réparti est de le partitionner en *grappes* distinctes, de sorte que chaque grappe soit un sous-ensemble connexe d'entités et qu'une de ces entités soit distinguée et nommée *tête de grappe*. Dans une telle *partition* d'un système réparti, chaque entité appartient à une *grappe* et obéit à la tête de sa grappe. Il existe différents types de partitionnements répartis [Bas99, BK01, FM02].

Nous en étudions un en particulier dans la partie III. Des définitions formelles de ce partitionnement y sont également détaillées.

Alliance. Les alliances d'entités ont pour but d'obtenir le contrôle global d'un système réparti à partir de la connaissance locale des entités. La définition d'alliance dans les graphes a été introduite par Kristiansen en 2004 [KHH04]. Une *alliance* est un sous-ensemble non vide d'entités tel qu'il assure, pour toute entité, certaines propriétés sur le nombre d'entités voisines qui sont dans l'alliance et le nombre de celles qui n'y sont pas. Une alliance est *défensive* si, pour chaque membre de l'alliance, il y a au moins autant de voisins alliés que de voisins non alliés. Sa construction est une généralisation de nombreux problèmes répartis. Par exemple, un *ensemble dominant* est une alliance défensive telle que chaque entité qui n'est pas dans l'ensemble dominant a au moins un voisin qui est dans l'ensemble dominant. Une bibliographie très complète de ce problème peut être trouvée dans [HL90]. La notion d'alliance défensive est très proche de celle de communauté Web, ainsi que le montre [FR07], qui est définie comme un ensemble d'entités tel que chaque membre a une majorité d'entités voisines qui en sont également membres. Nous étudions un cas particulier d'alliance dans la partie IV.

Sûreté de fonctionnement

Depuis le tout début des machines de calcul, la disponibilité, la fiabilité et la maintenabilité ont été des sujets constants de préoccupation. Ils sont englobés dans le concept de *sûreté de fonctionnement* proposé par Jean-Claude Laprie dans les années 1980 et qui peut-être intuitivement défini comme la « capacité à fournir un service en laquelle on peut légitimement faire confiance. » [ALRL04] Ce concept présente les caractéristiques suivantes : disponibilité, fiabilité, sûreté, confidentialité, intégrité et maintenabilité.

Dans cette thèse, nous nous intéressons plus particulièrement à la *fiabilité* des systèmes répartis et du service qu'ils fournissent. Par exemple, si un système réparti est consulté pour décider d'une valeur unique parmi un ensemble de valeurs réparties en amont du système, c'est-à-dire résoudre le problème du consensus, alors nous souhaitons que ce système fournisse en continu une valeur unique à tout moment. Il faut noter qu'il n'est pas nécessaire que ce soit toujours la même valeur.

D'une part, la sûreté de fonctionnement d'un système peut être amoindrie par les menaces qui pèsent sur chacune de ses caractéristiques. Quand un système est mis en marche, il fournit un service qui est conforme à sa spécification. Cependant, un événement, appelé *faute*, peut entraîner le système à avoir un comportement inattendu, et ainsi, à entrer dans un état non prévu. Les fautes seront décrites plus en détail dans la prochaine section. La différence entre l'état réel du système et son état attendu est appelé une *erreur*. Une telle erreur peut provoquer une *panne* du système vis-à-vis de sa spécification, c'est-à-dire, une incapacité à fournir le service demandé. Il est à noter qu'une panne peut à son tour déclencher une faute.

D'autre part, la sûreté de fonctionnement d'un système peut être améliorée grâce aux approches suivantes. De façon à légitimer la confiance dans la capacité d'un système à fournir un service, nous aimerions *prévoir* le nombre de fautes qui pourraient affecter le système et leur éventuelle incidence sur son comportement. Si ce nombre est trop élevé ou ne peut pas être évalué, nous pouvons alors essayer d'*éliminer* autant d'erreurs que possible du système. Et si nous ne pouvons pas toutes les éliminer, alors nous souhaiterions *prévenir* toutes les fautes restantes d'apparaître dans le système. Lorsque toutes les approches susmentionnées sont irréalisables, nous devons concevoir notre algorithme de telle sorte à éviter les pannes, même en présence de fautes, c'est-à-dire, un algorithme *tolérant aux fautes*. C'est particulièrement important pour les systèmes répartis à grande échelle, car ils sont particulièrement sujets aux dysfonctionnements matériels et ne peuvent pas se permettre de redémarrer leurs algorithmes à chaque fois qu'une faute apparaît.

Tolérance aux fautes

Nous caractérisons maintenant les fautes qui peuvent se produire dans un système par rapport à la durée de leur présence et à la fréquence de leurs apparitions. Certaines fautes, telles que le plantage définitif d'une entité ou la présence d'entités malveillantes [LSP82], sont *permanentes*. À l'opposé, la présence des fautes dites *temporaires* est bornée dans le temps. Quand des fautes temporaires apparaissent de façon répétitive à intervalles de temps réguliers, elles sont appelées *intermittentes*. C'est le cas, par exemple, d'une connexion peu fiable entre deux entités qui perdrait en moyenne un message sur deux. Enfin, les fautes temporaires qui se produisent rarement sont dites *transitoires*, comme par exemple la perte occasionnelle de message par une connexion qui serait d'ordinaire très fiable.

À partir de là, deux stratégies distinctes peuvent être adoptées pour tolérer les fautes, selon que les pannes sont inacceptables ou pas.

Dans le premier cas, il est exigé de l'algorithme qu'il suive impérativement sa spécification sans la moindre panne, même temporaire. Cela s'appelle la *robustesse*. À l'origine, c'était une des motivations sous-tendant le passage des systèmes centralisés aux architectures redondantes : La robustesse d'un seul système pouvait être améliorée en le répliquant. C'est ce qui a été fait dans la navette spatiale américaine [GS84]. Les algorithmes robustes peuvent masquer toutes les erreurs possibles dans l'état d'un système. Cette approche est dite *pessimiste* au sens où l'algorithme a une confiance extrêmement limitée dans la fiabilité du système sous-jacent. Cet évitement rigoureux des pannes n'est pas toujours possible à obtenir [FLP85] et a généralement un coût très élevé en termes de temps de calcul et d'espace mémoire.

Dans le second cas, certaines pannes peuvent être considérées comme acceptables tant qu'elles sont brèves et rares vis-à-vis de la durée de vie du système et des exigences de disponibilité de l'application. Une approche *optimiste* devient alors possible comme suit. Les erreurs ne sont pas nécessairement masquées à l'application et peuvent entraîner l'impossibilité pour le système de se conformer à la spécification

de haut niveau. L'algorithme doit alors gérer ces erreurs par lui-même pour que le système récupère un comportement conforme à sa spécification de haut niveau. Toutefois, les erreurs sont supposées être provoquées par des fautes transitoires uniquement. Le seul préalable ici est en fait qu'il doit y avoir des périodes de temps dénuées de fautes, de durée bien plus longue que le temps de récupération de l'algorithme. Cette approche permet des solutions efficaces dans un environnement hostile ou peu fiable dans un système réparti à grande échelle tel qu'Internet. L'auto-stabilisation, présentée ci-après, est une approche optimiste, sans masquage d'erreurs, à la tolérance aux fautes.

Auto-stabilisation

L'*auto-stabilisation* est une technique polyvalente pour tolérer toute faute transitoire dans un système réparti. Elle a été initialement introduite par Dijkstra en 1974 [Dij74] et mise en avant par Lamport en 1985 [Lam85]. De façon intuitive, un algorithme est auto-stabilisant si, après que des fautes transitoires touchent le système et le placent dans une configuration arbitraire, le système récupère sans intervention extérieure (ni manuelle) en temps fini. Ainsi, l'auto-stabilisation ne fait aucune hypothèse sur la nature ou l'étendue des fautes transitoires (à l'exception notable qu'elles ne peuvent pas toucher le code) et récupère des effets de ces fautes en une seule technique. À cette fin, les algorithmes auto-stabilisants ne reposent pas sur l'état initial du système. Cela facilite le déploiement d'un grand nombre d'entités, car les mémoires n'ont alors pas besoin d'être initialisée d'une manière particulière. Dans une certaine mesure, les algorithmes répartis auto-stabilisants sont aussi susceptibles de tolérer les changements topologiques *détectables* dans le contexte des réseaux dynamiques. Dans ce cas, il faut néanmoins supposer que les changements topologiques soient rares, comme pour les fautes transitoires. Certains algorithmes auto-stabilisants pour réseaux dynamiques ont été proposés par Dolev, Israeli et Moran en 1989 [DIM89].

Toutefois, l'auto-stabilisation présente quelques inconvénients. D'abord, la conception d'une solution auto-stabilisante à un problème réparti comporte généralement un coût supplémentaire comparé à une solution non auto-stabilisante. Ce coût peut se traduire par un temps d'exécution plus long ou des besoins en espace mémoire plus importants, voire les deux à la fois. Ensuite, l'auto-stabilisation consiste en une récupération globale d'un système réparti. Ainsi donc, les entités ne peuvent pas détecter localement si le système a déjà récupéré. Cela rend impossible la *détection de terminaison* des algorithmes auto-stabilisants pour les problèmes répartis. Enfin, le principal inconvénient est la *perte temporaire de sûreté*. Après l'apparition de fautes transitoires, il y a une période de temps de durée finie — appelée la *phase de stabilisation* — avant que le système ne récupère totalement. Pendant cette phase, il n'y a aucune garantie de sûreté, au sens où le système peut ne pas être conforme à sa spécification pendant ce temps. Plusieurs tentatives ont été faites pour mitiger ces inconvénients en proposant des alternatives à l'auto-stabilisation, qui seront résumées dans la prochaine section.

Propriétés dérivées

Nous considérons la notion de convergence sûre [KM06] dans la partie IV. L'idée principale derrière ce concept est la suivante : pour une grande classe de problèmes, il est souvent difficile de concevoir des algorithmes auto-stabilisants qui garantissent un temps de stabilisation court même après un faible nombre de fautes transitoires [GT02]. Un temps de stabilisation long est habituellement imputable aux spécifications fortes qu'une configuration légitime doit satisfaire. Le but d'un *algorithme auto-stabilisant avec convergence sûre* est de converger rapidement d'abord (généralement en $O(1)$ rondes) vers une configuration légitime *faisable*, où un service minimum est garanti. Une fois qu'une configuration légitime faisable est atteinte, le système continue de converger vers une configuration légitime *optimale*, où des conditions plus strictes doivent être remplies. La convergence sûre est particulièrement intéressante pour les algorithmes auto-stabilisants qui calculent des structures de données optimisées, telles qu'un ensemble dominant minimal [KM06], une approximation d'un ensemble dominant faiblement connexe minimum [KK07] ou d'un ensemble dominant connecté minimum [KK12].

Modèle

Système réparti et topologies

Dans cette thèse, un *système réparti* est défini comme un ensemble de n entités communicantes comme suit.

Une *entité* est une unité autonome de calcul, qui peut être modélisée par un automate déterministe. Chaque entité a un *identifiant unique*. L'ensemble des identifiants est totalement ordonné. À des fins d'analyse de complexité, l'hypothèse usuelle est faite que chaque identifiant est stocké sur $O(\log n)$ bits. Sauf indication contraire, les entités n'ont aucune connaissance globale *a priori* du système réparti. En particulier, elles ne connaissent ni n ni l'ensemble des identifiants.

Chaque entité est capable de *communiquer* avec un sous-ensemble d'autres entités. Deux entités distinctes capables de communiquer entre elles sont dites *voisines*. Les possibilités de communication entre deux entités voisines sont supposées être toujours *bidirectionnelles*. Chaque entité est capable de distinguer chacune de ses voisines en utilisant des noms locaux. L'ensemble des noms locaux à une entité p est noté $\mathcal{N}(p)$. Chaque entité connaît son propre nom local à chacun de ses voisins.

Graphe. La *topologie* du réseau de communication d'un système réparti est représentée par un *graphe* (simple et non orienté) $G = (V, E)$ où V est l'ensemble de ses entités et $E \subseteq V^2$ est l'ensemble des arêtes représentant les possibles communications directes entre entités voisines. Un *chemin* entre deux entités p et q est une séquence d'entités de V telle qu'il existe une arête de E entre chaque paire d'entités successives dans cette séquence. Un *cycle* est un chemin commençant et terminant par la même entité. La *longueur* d'un chemin est le nombre d'arêtes correspondantes. La *distance* entre deux entités est la longueur du plus court chemin qui les relie.

Le *diamètre* d'un graphe est la plus grande distance de toutes les paires d'entités. Le *degré* d'une entité est son nombre de voisins. Le *degré* d'un graphe est le plus grand degré de toutes les entités. Le graphe correspondant à un système est supposé *connexe*, c'est-à-dire que pour toute paire d'entités de ce système, il existe toujours au moins un chemin reliant ces deux entités dans ce graphe.

Arbre. Un *arbre* est un graphe connexe de $x - 1$ arêtes pour x entités. En particulier, il est *acyclique*. Une des entités de l'arbre est appelée *racine*. Le *niveau* d'une entité est sa distance à la racine dans l'arbre. La *hauteur* d'un arbre est le plus grand niveau de toutes ses entités. Le *parent* d'une entité non racine est son voisin sur le plus court chemin la reliant à la racine. Les *enfants* d'une entité sont ses voisins dont elle est le parent. Les *ascendants* d'une entité non racine sont toutes les entités sur le plus court chemin la reliant à la racine. Les *descendants* d'une entité sont toutes les entités dont elle est un ascendant. Les *feuilles* sont toutes les entités qui n'ont pas d'enfant. Un *arbre ordonné* est un arbre enraciné pour chaque entité non feuille duquel un ordre strict total est localement défini sur l'ensemble de ses enfants. Un *arbre couvrant* d'un graphe $G = (V, E)$ de n entités est un sous-graphe $T = (V_T, E_T)$ tel que $V_T = V$, $E_T \subseteq E$ et E_T compte $n - 1$ arêtes.

Communication par mémoires localement partagées

Nous adoptons les *variables localement partagées* introduites par Dijkstra [Dij74], où chaque entité possède un ensemble fini de variables, permettant de réaliser la communication comme suit : Chaque entité peut lire ses propres variables et les variables de ses voisins, mais elle ne peut écrire que ses propres variables.

Programme local. Le *programme local* d'une entité p est défini par un ensemble fini d'*actions* (ou commandes gardées), chacune composées de : (1) un nom, (2) une *garde* qui est une expression booléenne pouvant impliquer les variables de p et de ses voisins, (3) une *instruction* qui est une séquence d'affectations mettant à jour une ou plusieurs variables de p. Nous adoptons également le modèle d'*atomicité composite* [DIM93], c'est-à-dire que l'évaluation des gardes et l'exécution des instructions est supposée avoir lieu en un seul pas atomique (ou ininterrompu).

Une action peut être exécutée si et seulement si sa garde est évaluée comme vrai, nous disons alors que cette action est *activable*. Par extension, une entité est *activable* si et seulement si au moins une de ses actions est activable. Dans l'écriture de nos programmes locaux, les actions d'une entité sont exclusives entre elles.

Algorithme réparti

Un *algorithme réparti* est une collection de n programmes locaux, chacun d'eux opérant sur une entité distincte. Il faut noter qu'un algorithme réparti peut être conçu pour un ensemble restreint de topologies $\mathcal{G}$. Soit $\mathcal{A}$ un algorithme réparti. Pour toute entité p, nous notons $\mathcal{A}(p)$ le programme local de p dans $\mathcal{A}$.

État et configuration. L'*état* d'une entité p dans le programme local $\mathcal{A}(p)$ est défini par les valeurs de ses variables dans $\mathcal{A}(p)$. Une *configuration* est une instance des états de toutes les entités dans $\mathcal{A}$. Une configuration est *terminale* si et seulement si aucune entité n'est activable dans cette configuration. L'ensemble de toutes les configurations possibles est noté $\mathcal{C}$. Soit $\gamma \in \mathcal{C}$, alors $\gamma(p)$ représente l'état local de l'entité p dans la configuration γ.

Pas. Soit γ une configuration, nous notons l'ensemble des entités activables dans γ par $V_{ac}(\gamma) \subseteq V$. Si $V_{ac}(\gamma) = \emptyset$, alors γ est terminal. Sinon un sous-ensemble non vide d'entités de $V_{ac}(\gamma)$ est *activé* et noté $V_{ex}(\gamma)$. Chaque entité de $V_{ex}(\gamma)$ exécute son action activable, menant ainsi à une nouvelle configuration γ'. Une telle transition de γ à γ' est appelée un *pas* et notée $\gamma \mapsto \gamma'$. Notons que $(\mathcal{C}, \mapsto)$ est un système de transition des configurations de $\mathcal{A}$.

Exécution. Une *exécution* de $\mathcal{A}$ est une séquence maximale de configurations $e = \gamma_0\gamma_1 \dots \gamma_i \dots$ telle que $\gamma_{i-1} \mapsto \gamma_i$ pour tout $i > 0$. « Maximale » s'entend au sens où l'exécution est infinie ou s'arrête par une configuration terminale.

L'ensemble $V_{ac}(\gamma)$ des entités activables dans une configuration γ est calculé en évaluant les gardes de $\mathcal{A}$. L'ensemble $V_{ex}(\gamma)$ des entités activées dans la configuration γ est sélectionné par un ordonnanceur défini ci-après. L'ordonnanceur matérialise l'asynchronisme du système : chaque entité peut travailler à différentes vitesses dans le temps et indépendamment des autres.

Ordonnanceur. Soit $\mathcal{A}$ un algorithme réparti et $\mathcal{G}$ un ensemble de topologies.

Nous supposons que chaque pas $\gamma \mapsto \gamma'$, d'une configuration de $\mathcal{C}$ à une autre, est déterminé par un *ordonnanceur*, c'est-à-dire un adversaire qui sélectionne l'ensemble $V_{ex}(\gamma)$ des entités qui sont activées, comme un sous-ensemble de $V_{ac}(\gamma)$.

Cet ordonnanceur est dit *propre*, car il sélectionne nécessairement, à chaque pas, un sous-ensemble non-vide de $V_{ac}(\gamma)$ dans le cas où celui-ci est non-vide. Cette *propriété de progrès* garantit ainsi que l'ordonnanceur ne peut pas complètement empêcher l'exécution de l'algorithme. En l'absence d'hypothèse supplémentaire, un tel ordonnanceur est aussi dit *inéquitable*, car il peut tout de même empêcher une entité d'être activée aussi longtemps qu'elle n'est pas la seule entité activable. Un ordonnanceur est dit *faiblement équitable* si et seulement si il finit toujours par activer toute entité continument activable.

Silence. Un algorithme réparti est *silencieux* si chacune de ses exécutions est finie. Autrement dit, en partant d'une configuration quelconque, le système finira nécessairement par atteindre une configuration terminale.

Complexité en espace. La complexité en espace (mémoire) d'un algorithme réparti $\mathcal{A}$ est calculée à partir du nombre de bits nécessaire pour représenter tout état possible de $\mathcal{A}(p)$ pour toute entité p.

Complexité en temps. Pour évaluer la complexité d'un algorithme réparti $\mathcal{A}$, c'est-à-dire le temps qu'il requiert pour atteindre une certaine configuration, nous disposons de deux unités de temps que sont le pas (défini ci-avant) et la ronde (définie ci-après). Intuitivement, une ronde est le temps mis par l'entité la plus lente pour progresser dans $\mathcal{A}$. Une entité p est *neutralisée* dans le pas $\gamma_i \mapsto \gamma i + 1$ si p est activable dans γ_i et non activable dans γ_{i+1} sans avoir exécuté la moindre action entre ces deux configurations. Une entité ne peut être neutralisée si un ou plusieurs de ses voisins ont changé leur état pendant ce pas. Enfin, nous définissons la notion de *ronde* de façon inductive comme suit. Soit e une exécution. La première ronde de e, notée e', correspond au plus petit préfixe de e dans lequel chaque entité activable dans la configuration initiale exécute une action ou est neutralisée. Soit e'' un suffixe de e commençant par la dernière configuration de e'. La deuxième ronde de e est la première ronde de e'', etc.

Auto-stabilisation

Soit $\mathcal{A}$ un algorithme réparti, $\mathcal{G}$ un ensemble de topologies et $\mathfrak{D}$ une famille d'ordonnanceurs. Soit un prédicat P sur toutes les configurations de $\mathcal{A}$ dans toute instance de $\mathcal{G}$. Toute configuration où P est satisfait est dite *légitime*. Alors $\mathcal{A}$ est une algorithme *auto-stabilisant* pour le prédicat P si les trois conditions suivantes sont remplies pour toute instance de $\mathcal{G}$ et toute instance de $\mathfrak{D}$: (1) il existe au moins une configuration légitime ; (2) le prédicat P est clos pour les actions de $\mathcal{A}$; (3) $\mathcal{A}$ fait converger P vers vrai, c'est-à-dire que toute exécution de $\mathcal{A}$ contient une configuration légitime. Le *temps de stabilisation* de $\mathcal{A}$ est le temps maximum requis pour atteindre une configuration légitime depuis n'importe quelle configuration de $\mathcal{C}$.

Auto-stabilisation avec convergence sûre. La convergence sûre est une propriété complémentaire à l'auto-stabilisation dans le but d'améliorer la sûreté pendant la phase de stabilisation. Soit un algorithme réparti $\mathcal{A}$, $\mathcal{G}$ un ensemble de topologies, $\mathfrak{D}$ une famille d'ordonnanceurs et (P_1, P_2) une paire de prédicats sur les configurations de $\mathcal{A}$ pour toute instance de $\mathcal{G}$. Alors $\mathcal{A}$ est *auto-stabilisant avec convergence sûre pour* (P_1, P_2) si et seulement si il respecte les deux points suivants : (1) $\mathcal{A}$ stabilise (rapidement) d'une configuration quelconque vers une configuration où le prédicat réalisable P_1 est satisfait ; (2) à partir d'une telle configuration, $\mathcal{A}$ stabilise ensuite, vers une configuration où le prédicat optimum P_2 est satisfait. Il faut noter que P_1 est clos pour les actions de $\mathcal{A}$, donc que P_1 est satisfait pendant toute la deuxième partie de la phase de stabilisation, d'où le nom de convergence sûre. Cette propriété est particulièrement adaptée aux problèmes de construction de structures pouvant être optimisées [KM06, KK07, KK12].

Composition collatérale hiérarchique

Les techniques de composition d'algorithmes sont fréquemment utilisées pour simplifier la conception et les preuves des algorithmes auto-stabilisants [Tel01]. La

composition collatérale hiérarchique proposée dans cette thèse est une variante de la composition collatérale [Her92]. Elle permet d'exécuter deux algorithmes de façon concurrente, le second algorithme utilisant la sortie du premier dans ses calculs. Ici, l'exécution des algorithmes est de plus hiérarchisée en contraignant toute entité à n'exécuter une action du deuxième algorithme que si aucune action du premier algorithme n'est exécutable. En cela, elle diffère également de la composition équitable [Dol00].

Soient $\mathcal{A}$ et $\mathcal{B}$ deux algorithmes auto-stabilisants en présence d'un ordonnanceur faiblement équitable, tels que $\mathcal{B}$ peut lire les variables de $\mathcal{A}$ et qu'aucun des deux algorithmes ne peut modifier les variables de l'autre. Notons $\mathcal{B} \circ \mathcal{A}$ la composition collatérale hiérarchique de ces deux algorithmes. Alors $\mathcal{B} \circ \mathcal{A}$ stabilise en vérifiant la spécification *Spec* en présence d'un ordonnanceur faiblement équitable si $\mathcal{A}$ est silencieux et $\mathcal{B}$ stabilise en vérifiant la spécification *Spec* à partir de toute configuration où toutes les actions de $\mathcal{A}$ sont désactivées pour toujours. (Notons que dans une telle configuration, la spécification de $\mathcal{A}$ est satisfaite.)

Transformation d'algorithme pour l'équité

Une méthode automatique est donnée dans cette thèse pour transformer tout algorithme auto-stabilisant fonctionnant en présence d'un ordonnanceur faiblement équitable en algorithme auto-stabilisant fonctionnant en présence d'un ordonnanceur inéquitable, pour la même spécification. Elle consiste en une composition avec un algorithme d'horloge de phase, tel que celui proposé par Boulinier [BPV04]. Notons R la complexité en rondes de l'algorithme d'entrée. Cette méthode produit un algorithme de sortie de complexité R en rondes et $O(\mathcal{D}n(R + n^2))$ en pas de calcul. De plus, cette méthode préserve la propriété de silence. Bien qu'il existe déjà de nombreuses méthodes pour passer de l'équité faible à l'inéquité [BGJ01, KK06], aucune d'entre elles offre à la fois la préservation de la propriété de silence et une aussi faible dégradation de la complexité de l'algorithme de sortie. Cette méthode de transformation peut donc être avantageusement appliquée à tous les algorithmes proposés dans cette thèse, afin de relâcher l'hypothèse d'équité faible de l'ordonnanceur.

Partie II : Structures de soutien

Dans cette partie, nous présentons deux structures de données réparties dont nous nous servons dans la suite de cette thèse. Bien que ce soit la raison première de leur étude ici, elles offrent aussi un intérêt plus général, c'est pourquoi elles sont présentées séparément de leur application. Nous abordons d'abord l'arbre couvrant à ensemble maximal indépendant, et les paires de guidage ensuite. Pour chacune d'entre elles, nous présentons un algorithme pour la construire, prouvons qu'il est auto-stabilisant dans notre modèle et analysons sa complexité en temps et en espace. De plus nous identifions la classe de complexité du problème résolu par l'algorithme construisant un arbre couvrant à ensemble maximal indépendant.

Arbre couvrant à ensemble indépendant maximal

Nous nous intéressons d'abord à l'*arbre couvrant EIM* (pour ensemble indépendant maximal) initialement introduit par Fernandess et Malkhi [FM02] et à sa construction auto-stabilisante. Soit un graphe connexe G. Un arbre couvrant EIM de G est un arbre couvrant T de G enraciné à un sommet r tel que l'ensemble P des sommets des niveaux pairs de T est un ensemble indépendant maximal de G. Un ensemble S de sommets est *indépendant* dans G si et seulement si aucun des sommets de S n'est voisin d'un autre sommet de S dans G. Un ensemble indépendant peut être dit *maximal* pour l'inclusion. Intuitivement, il faut noter que tout chemin dans T est constitué pour moitié de sommets de P qui ne sont pas voisins entre eux dans G. Nous exploitons cette propriété en particulier dans notre algorithme de k-partitionnement compétitif proposé dans la partie III.

Contribution. Nous proposons ici un algorithme auto-stabilisant et silencieux qui construit un arbre couvrant EIM en $O(n)$ rondes en présence d'un ordonnanceur faiblement équitable, où n est le nombre d'entités du système. De plus, nous montrons que la hauteur de l'arbre couvrant EIM fait, au plus, le double du diamètre du graphe pour lequel il est calculé par cet algorithme. Nous prouvons enfin que le problème spécifique qu'il résout en temps linéaire est $\mathcal{P}$-complet. Selon l'hypothèse majoritairement soutenue que $\mathcal{NC} \neq \mathcal{P}$, ce problème serait donc « intrinsèquement séquentiel ». Ceci semble nous indiquer qu'il est très difficile, voire impossible, de trouver une solution qui stabilise en temps sous-linéaire exprimé en rondes.

Algorithme. Il se présente sous la forme d'une composition collatérale hiérarchique de deux algorithmes auto-stabilisants et silencieux. Le premier construit un arbre couvrant classique et peut être instancié par un des nombreux algorithmes existants dans la littérature [CYH91, HC92, CD94, DLV11a]. Le second construit un arbre couvrant EIM en utilisant, pour chaque entité, sa hauteur dans l'arbre calculé par le premier algorithme.

Ce dernier algorithme définit, pour chaque entité, un booléen indiquant si l'entité appartient à l'ensemble indépendant maximal (à terme) et un pointeur indiquant parmi ses voisins le père de cette entité dans l'arbre couvrant EIM (à terme).

Pour déterminer quelles entités appartiennent à l'EIM, une priorité est définie pour les entités par plus petite hauteur d'abord, puis par plus petite identité. Chaque entité décide de faire partie de l'EIM si et seulement si tous ses voisins ne sont pas dans l'EIM ou sont moins prioritaires. En suivant cette règle, la racine de l'arbre calculé par le premier algorithme étant la plus prioritaire, elle sera nécessairement dans l'EIM, et donc tous ses voisins en seront exclus ; parmi les entités restantes, la suivante par priorité sera dans l'EIM, *etc.* Ainsi, on s'assure que l'ensemble calculé est indépendant et maximal.

Enfin, chaque entité détermine son père dans l'arbre EIM comme étant son voisin le plus prioritaire et dans un ensemble différent du sien (parmi l'EIM et son complémentaire). La racine des deux arbres étant la même, et l'alternance entre

les ensembles étant respectée entre chaque entité et son père, il en résulte que la structure couvrante calculée est bien un arbre couvrant EIM.

Étiquetage dans les arbres ordonnés

Nous nous intéressons maintenant à un étiquetage dans les systèmes répartis dont la topologie de réseau est un arbre ordonné, calculant pour chaque entité un index particulier appelé *paire de guidage*. Pour chaque entité p, sa paire de guidage est composée du rang de p dans le *parcours préfixe* de l'arbre ordonné et du rang de p dans le *parcours suffixe inverse* de l'arbre ordonné. En comparant deux paires de guidage préalablement calculées, il est possible de déterminer si l'une des deux entités correspondantes est une descendante de l'autre. Ce schéma d'étiquetage a de nombreuses applications [FEP+06] telles que le parcours ordonné et la navigation dans les réseaux en arbre. Nous l'utilisons ainsi dans la partie V afin de faire naviguer des paquets entre la racine et les autres entités d'un système réparti.

Travaux Connexes. La notion de *paires de guidage* est apparue pour la première fois dans [FEP+06, page 702] où elle est utilisée comme entrée d'un autre algorithme. Un algorithme auto-stabilisant pour réseau en arbre est donné dans [CT05b] qui calcule, pour chaque entité, son rang dans plusieurs parcours d'un arbre, y compris le parcours préfixe et le parcours suffixe inverse. Les auteurs montrent que leur algorithme stabilise en $O(n)$ rondes en présence d'un ordonnanceur central. Notons que l'algorithme que nous présentons dans ce chapitre est une instanciation de l'approche générale de [CT05b], cependant nous ne faisons pas l'hypothèse d'un ordonnanceur central.

Contribution. Nous prouvons dans cette thèse que cet algorithme est auto-stabilisant pour tout système réparti dont la topologie de réseau est un arbre ordonné, sous l'hypothèse d'un ordonnanceur réparti faiblement équitable. Ainsi l'algorithme stabilise en $O(h)$ rondes, où h est la hauteur de l'arbre ordonné, et requiert $O(\delta(p) \log n)$ bits par entité p, où n est le nombre d'entités dans le système.

Partie III : k-Partitionnement

Dans cette partie, nous considérons le problème de *k-partitionnement* défini ci-après. Trouver une solution optimum à ce problème est connu pour être $\mathcal{NP}$-difficile [GJ79]. À ce problème nous proposons deux approches originales diamétralement opposées. Enfin nous concluons cette partie en comparant ces deux approches par simulation.

Définitions

Soit un graphe connexe $G = (V, E)$, une *grappe* de G est définie par un sous-ensemble non vide de V et un sommet dit de *tête* appartenant à ce sous-ensemble. Soit un entier naturel k, une *k-grappe* de G est une grappe dont chaque sommet est à

distance au plus k du sommet de tête. Une *k-partition* de G est une partition de V en k-grappes de G. La *taille* d'une k-partition est son nombre de k-grappes.

Le problème de *k-partitionnement* est, pour k un entier naturel et G un graphe, de construire une k-partition de G. Une solution *optimum* à ce problème est donc une k-partition ayant le plus petit nombre de k-grappes possible.

Notons qu'il existe plusieurs homonymes dans la littérature qui doivent être clairement distingués. Par exemple, une partition d'un graphe en k parties distinctes est aussi parfois [Bru78] et [OR00] appelée « k-partition », ou « k-clustering » en anglais.

Un algorithme de k-partitionnement est dit *compétitif de ratio α* [FM02] s'il calcule une k-partition de taille au plus α fois celle d'une k-partition la plus petite possible.

Rapports entre k-partition et ensemble k-dominant

Soit un graphe $G = (V, E)$ et un entier naturel k, un *ensemble k-dominant* est un sous-ensemble de sommets D de V tel que chaque sommet de $V \setminus D$ est à distance au plus k d'un sommet de D.

Là encore, notons qu'il existe plusieurs homonymes dans la littérature qui ne doivent pas être confondus. Par exemple, un ensemble dominant k-redondant [KK03], qui est un sous-ensemble R de V tel que tout sommet de $V \setminus R$ a au moins k sommets de R parmi ses voisins (c'est-à-dire à distance 1), est aussi parfois [HLCW07, WWTZ12] appelé ensemble k-dominant, ou « k-dominating set » en anglais.

Remarquons que l'ensemble des têtes de k-grappes d'une k-partition est k-dominant. Il est donc trivial de déduire un ensemble k-dominant d'une k-partition. Réciproquement, il est possible de construire une k-partition à partir d'un ensemble k-dominant, en utilisant chaque membre de cet ensemble comme tête d'une k-grappe et en attribuant une tête de k-grappe à chaque entité qui n'est pas cet ensemble. Il suffit pour cela de choisir la tête de k-grappe la plus proche, c'est ce que fait notamment l'algorithme auto-stabilisant de Datta, Devismes et Larmore [DDL09].

Travaux connexes

Il existe de nombreux algorithmes répartis auto-stabilisants construisant une k-partition [CDDL10, DLV09, DDL09] ou un ensemble 1-dominant minimal [SRR95, IKK02]. Aucun d'entre eux ne s'intéresse à la compétitivité. Il existe également des solutions *non auto-stabilisantes* pour construire un k-partition [APHV00, FM02, SGLA04, Rav05]. Parmi celles-ci, seule celle de Fernandess et Malkhi [FM02] s'intéresse à la compétitivité.

Le problème de construire de façon déterministe un ensemble k-dominant de $\lceil \frac{n}{k+1} \rceil$ entités au plus a été étudié par Peleg et Upfal [PU89]. Leur solution suppose que le système est synchrone (c'est-à-dire qu'à chaque pas toutes les entités activables exécutent une action). Les auteurs ont omis un cas particulier qui invalide malencontreusement leur preuve pour certains réseaux. La même faille est présente

dans plusieurs travaux qui en découlent [KP95a, PB04].

Enfin, Ravelomanana [Rav05] donne un algorithme ayant recours aux tirages aléatoires pour construire une k-partition dans des réseaux synchrones dont la topologie est un UDG. Cet algorithme converge en $O(\mathcal{D})$ rondes, où $\mathcal{D}$ est le diamètre du réseau.

Construction d'un ensemble k-dominant minimal borné

Cette première approche envisage de construire un ensemble k-dominant minimal pour l'inclusion et de taille bornée. Il faut remarquer qu'un ensemble k-dominant minimal n'est pas nécessairement de petite taille. Il semble donc utile de garantir une borne supérieure sur la taille de l'ensemble k-dominant que l'on souhaite construire.

Contribution. Ce travail a été initié à partir d'une borne supérieure trouvée par Peleg et Upfal [PU89], majorant la taille de tout ensemble k-dominant minimum. Nous montrons dans cette thèse que leur preuve constructive de cette borne contient une erreur. Nous proposons alors un correctif ne modifiant pas la borne. À partir de cette nouvelle preuve, nous construisons un algorithme distribué, asynchrone, silencieux et auto-stabilisant calculant un ensemble k-dominant minimal contenant au plus $\lceil \frac{n}{k+1} \rceil$ entités d'un système réparti quelconque, où n est le nombre d'entités du système. Notre algorithme est prouvé en supposant un ordonnanceur faiblement équitable, il stabilise en $O(n)$ rondes et nécessite $O(\log k + \log n + k \log \frac{N}{k})$ bits par entité, où N est un majorant de n. Il faut noter que N est introduit uniquement pour borner les exigences en mémoire de l'algorithme. Ce faisant, nous supposons que chaque entité connait une valeur de N, sans avoir besoin de connaître la valeur de n.

Algorithme général. Il se présente sous la forme d'une composition collatérale hiérarchique de trois algorithmes auto-stabilisants silencieux.

- Le premier algorithme doit construire un arbre couvrant enraciné. Il peut être instancié par un algorithme de Datta, Larmore et Vemula [DLV11a].
- En utilisant cette structure d'arbre couvrant, le deuxième algorithme, détaillé ci-après, construit un ensemble k-dominant borné. Nous montrons en effet que cet ensemble contient au plus $\lceil \frac{n}{k+1} \rceil$ entités. Cette construction correspond au schéma de la preuve corrigée de la borne supérieure sur la taille d'un ensemble k-dominant minimum.
- Le troisième et dernier algorithme doit rendre minimal pour l'inclusion l'ensemble k-dominant borné calculé par le deuxième algorithme. Pour ce faire, nous pouvons utiliser un algorithme proposé par Datta, Devismes et Larmore [DDL09].

Construction d'un ensemble k-dominant borné dans un arbre. Elle se déroule en trois phases successives, comme suit.

1. *Coloriage.* Chaque entité maintient une valeur dans $[0..k]$ appelée *couleur*. L'entité à la racine de l'arbre étant de hauteur 0, elle s'assigne nécessairement la couleur 0. Chaque enfant de la racine en déduit que sa couleur est 1 en faisant la somme modulo $k+1$ de 0 (couleur du parent) et de 1. Le calcul se poursuit ainsi en descendant l'arbre jusqu'à ses feuilles.

2. *Comptage.* Chaque entité maintient un compte du nombre d'entités par couleur dans son sous-arbre. Les feuilles de l'arbre mettent donc le compte de leur propre couleur à 1 et celui de toutes les autres couleurs à 0. Chaque entité non feuille fait la somme des comptes de ses enfants et ajoute 1 pour le compte de sa propre couleur. Les totaux remontent ainsi dans l'arbre jusqu'à la racine.

3. *Sélection.* Chaque entité maintient une valeur supplémentaire dans $[0..k]$ correspondant à la couleur choisie pour constituer, avec la racine, l'ensemble k-dominant. L'entité à la racine de l'arbre décide de cette valeur en fonction des totaux obtenus. Cette valeur est ensuite propagée, de parent en enfants jusqu'aux feuilles de l'arbre, à toutes les entités du système.

L'ensemble ainsi construit au terme de ces trois phases est bien k-dominant car il vérifie une propriété plus forte encore : toute entité qui n'est pas dans cet ensemble a un ascendant à distance au plus k qui est dans cet ensemble. Et comme les couleurs partitionnent les n entités du système en $k+1$ ensembles disjoints, le plus petit de ces ensembles, en y ajoutant la racine, est ainsi fait d'au plus $\lceil \frac{n}{k+1} \rceil$ entités.

k-Partitionnement compétitif

Cette seconde approche vise à construire, pour tout système, une k-partition dont la taille approxime celle d'une k-partition la plus petite possible pour le même système. En cela, elle diffère déjà de l'approche précédente qui se contentait d'une borne supérieure sur la taille d'une k-partition la plus petite possible pour tout système.

Contribution. Cet algorithme a été élaboré après l'algorithme précédent construisant un ensemble k-dominant et en est pratiquement le contrepied. Nous proposons ici un algorithme auto-stabilisant et silencieux qui construit une k-partition de n'importe quel système. Notre algorithme est prouvé en supposant un ordonnanceur faiblement équitable, stabilise en $O(n)$ rondes et nécessite $O(\log k + \log n)$ bits par entité où n est le nombre d'entités du système.

Dans le cas général, notre algorithme construit au plus $\lceil \frac{n}{k+1} \rceil$ k-grappes. Dans le cas où le réseau est un UDG, notre algorithme est compétitif de ratio $7.2552k + O(1)$. Dans le cas plus général où le réseau est un ADG avec un ratio d'approximation λ, notre algorithme est compétitif de ratio $7.2552\lambda^2 k + O(\lambda)$. Enfin, dans le cas d'un réseau en arbre, notre algorithme calcule une k-partition ayant le nombre minimum de k-grappes.

Composition. Comme pour l'algorithme précédent, cet algorithme requiert une structure d'arbre pour réaliser sa construction. Pour fonctionner dans tout système, il suffit encore une fois d'avoir recours à une composition collatérale hiérarchique avec un algorithme construisant un arbre couvrant.

Construction d'une k-partition optimum dans un arbre. Chaque entité calcule une valeur α représentant sa « distance à son descendant le plus éloigné qui soit dans la même k-grappe ». Cette distance est trivialement nulle pour chaque entité feuille. Elle croît progressivement en remontant dans l'arbre. Lorsqu'elle atteint k pour une entité p, celle-ci est désignée comme tête de k-grappe. Cette distance peut encore être incrémentée en remontant dans l'arbre jusqu'à atteindre la valeur $2k$. Au delà, les entités sont à une distance plus grande que k de p, donc elles ne peuvent donc pas être dans la même k-grappe que p. La valeur α peut alors repartir de 0 pour l'entité suivante.

La dernière k-grappe à se constituer contient la racine qui décidera d'en prendre la tête si nécessaire. Toutes les autres k-grappes contiennent au moins un chemin de longueur k. Cette propriété permet de vérifier la borne supérieure précédemment démontrée sur la taille de la k-partition construite.

En comparaison avec l'algorithme précédent, la valeur calculée par chaque entité est dans l'intervalle $[0..2k]$ au lieu de $[0..k]$. De plus, et à la grande différence de l'algorithme précédent, le sens de calcul est effectué des feuilles vers la racine. Alors que les marges se trouvent dispersées autour des feuilles dans l'algorithme précédent, elles sont ici localisées à la racine. En définitive, on s'attend à ce que l'espacement entre les têtes de k-grappes soit plus grand et que leur nombre soit plus petit.

Compétitivité dans les UDG et les ADG. La topologie des réseaux de capteurs sans fil est contrainte par la position physique des capteurs et la portée de transmission de leur radio. Les deux types de graphes suivants sont souvent utilisés pour la modéliser. Dans un *graphe de disques unitaires* (UDG), toute paire de sommets est directement connectée si et seulement si la distance euclidienne entre ces deux sommets est inférieure ou égale à 1. Plus généralement, dans un *graphe de disques unitaires approchés* (ADG) de *ratio* λ, pour toute paire de sommets, non seulement ces sommets sont connectés si la distance euclidienne entre eux est inférieure ou égale à 1, mais en plus ces sommets peuvent possiblement être connectés si cette distance est inférieure ou égale à λ.

En composant cet algorithme avec la construction d'un arbre couvrant EIM présentée dans la partie II, on obtient que chaque k-grappe (sauf celle de la racine) contient au moins $\lceil \frac{k}{2} \rceil$ entités de l'ensemble indépendant maximal constitué des niveaux pairs de l'arbre couvrant EIM. En utilisant un résultat de géométrie de Folkman et Graham [FG69], on peut borner la taille de cet ensemble en fonction de k et de la taille minimum d'une k-partition, puisque la distance euclidienne entre deux entités indépendantes est supérieure à 1 dans le cas d'un UDG, respectivement $1 + \lambda$ dans le cas d'un ADG. Dès lors, on peut établir un rapport entre le nombre de

k-grappes construites par l'algorithme et la taille minimum d'une k-partition.

Simulations

Nous avons souhaité évaluer nos deux algorithmes présentés ci-avant au-delà des cas limites. Nous avons donc réalisé des simulations pour étudier leur performance moyenne en nombres de têtes k-grappes obtenues.

Protocole expérimental. Nous avons utilisé un simulateur événementiel de réseaux sans fil nommé *Sinalgo* [Flu08]. Nous avons considéré des systèmes dont la topologie est un UDG composé de 1000 entités distribuées aléatoirement selon une loi uniforme dans un plan carré. Nous avons fait varier le paramètre k de 1 à 5 et le degré moyen de 10 à 30. Pour chaque réglage, nous avons aléatoirement généré 50 réseaux et nous avons compté le nombre moyen de têtes de k-grappes obtenues sur ces réseaux par l'algorithme à évaluer.

Algorithmes comparés. Après avoir essayé plusieurs algorithmes pour construire un arbre couvrant [DDH+11a], nous avons retenu celui de Huang et Chen [HC92] qui effectue une recherche en largeur d'abord dans le réseau. Nous l'avons utilisé comme élément de base dans la composition de nos deux principaux algorithmes : la construction d'un ensemble k-dominant de taille bornée et le k-partitionnement compétitif. Nous avons alors comparé ces deux algorithmes entre eux. Puis nous avons ajouté l'algorithme de k-partitionnement minimal de Datta, Devismes et Larmore [DDL09] dans nos simulations. Ce sont les seuls algorithmes connus pour garantir une borne supérieure intéressante sur le nombre de k-partitions obtenues.

Analyse des résultats. Nous avons d'abord observé que l'algorithme de k-partitionnement compétitif donne des résultats bien meilleurs que ceux de l'algorithme construisant un ensemble k-dominant borné. En les composant avec l'algorithme de k-partitionnement minimal, nous avons aussi constaté une baisse très importante de la taille des k-partitions calculées. Enfin, en comparant ces compositions avec l'algorithme de k-partitionnement minimal seul, nous avons constaté que les résultats sont très proches, néanmoins l'algorithme de k-partitionnement compétitif avec minimalisation reste le plus performant.

Partie IV : (f, g)-Alliance avec convergence sûre

Dans cette partie, nous considérons le *problème de* (f, g)*-alliance* qui consiste à construire une (f, g)-alliance dans un système réparti telle que définie ci-après. Une (f, g)-alliance est une généralisation de nombreuses structures couvrantes réparties qui présentent un certain intérêt pour le domaine des systèmes répartis. Nous donnons ici un algorithme auto-stabilisant silencieux avec convergence sûre qui calcule une (f, g)*-alliance minimale* en présence d'un ordonnanceur inéquitable.

Définitions

Soit $G = (V, E)$ un graphe non orienté et soient f et g deux fonctions associant à chaque entité p de V une image dans l'ensemble des entiers naturels. Un sous-ensemble d'entités $A \subseteq V$ est une (f, g)-*alliance* de G si et seulement si :

$$(\forall p \in V \setminus A, |\mathcal{N}(p) \cap A| \geq f(p)) \wedge (\forall p \in A, |\mathcal{N}(p) \cap A| \geq g(p))$$

À cette définition viennent s'ajouter deux notions de minimalité :

- A est une (f, g)-*alliance minimale* si et seulement si il n'existe pas de sous-ensemble propre de A qui soit une (f, g)-alliance de G ;
- A est une (f, g)-*alliance* 1-*minimale* si et seulement si $\forall p \in A$, $A \setminus \{p\}$ n'est pas une (f, g)-alliance de G.

Il faut noter que toute (f, g)-alliance minimale est trivialement une (f, g)-alliance 1-minimale, mais que la réciproque n'est pas toujours vraie. Dourado *et al.* ont néanmoins montré que si $f(p) \geq g(p)$ pour toute entité p, alors toute (f, g)-alliance 1-minimale est une (f, g)-alliance minimale [DPRS11].

Liens avec d'autres structures de données

Une (f, g)-alliance est une généralisation de nombreuses structures de données couvrantes qui présentent un certain intérêt dans le domaine des systèmes répartis. Considérons un sous-ensemble S quelconque d'entités :

1. S est un ensemble dominant (minimal) si et seulement si S est une $(1, 0)$-alliance (minimale) ;
2. plus généralement, S est un ensemble k-redondant dominant (minimal) si et seulement si S est une $(k, 0)$-alliance (minimale) ;
3. S est un ensemble k-tuple dominant (minimal) si et seulement si S est une $(k, k-1)$-alliance (minimale);
4. S est une alliance défensive globale (minimale) si et seulement si S est une $(f, 0)$-alliance (minimale), telle que $\forall p \in V$, $f(p) = \left\lceil \frac{\delta(p)}{2} \right\rceil$;
5. S est une alliance offensive globale (minimale) si et seulement si S est une $(1, g)$-alliance (minimale), telle que $\forall p \in V$, $g(p) = \left\lceil \frac{\delta(p)}{2} \right\rceil$.

Notons que les (f, g)-alliances ont aussi des applications dans le domaine des protocoles de population [AAER07] ou encore pour l'allocation de serveur dans les réseaux d'ordinateurs [GMOR05].

Contribution

Nous donnons un algorithme auto-stabilisant silencieux, $\mathcal{MA}(f, g)$, qui calcule une (f, g)-alliance minimale dans un système dont la topologie de réseau $G = (V, E)$ est non orientée, en présence d'un ordonnanceur inéquitable, où f et g sont deux fonctions, de l'ensemble des entités vers celui des entiers naturels, définies telles que pour toute entité p, $f(p) \geq g(p)$ et $\delta(p) \geq g(p)$, où $\delta(p)$ est le nombre de voisins de p. La première hypothèse sur f et g nous place dans le cas où toute (f, g)-alliance 1-minimale est une (f, g)-alliance minimale. La seconde hypothèse sur g nous garantit qu'une (f, g)-alliance est toujours possible.

Nous remarquons que la classe des (f, g)-alliances minimales telles que $\forall p \in V, f(p) \geq g(p)$ généralise les classes des ensembles dominants minimaux, des ensembles k-redondants dominants minimaux, des ensembles k-tuples dominants minimaux et des alliances offensives globales minimales. Par contre, les alliances défensives globales minimales ne sont pas inclues dans cette classe.

Notre algorithme $\mathcal{MA}(f, g)$ est auto-stabilisant *avec convergence sûre* dans le sens où, en partant d'une configuration quelconque, il calcule d'abord une (f, g)-alliance (pas nécessairement minimale) en quatre rondes au plus, puis il continue à faire converger l'état du système vers une (f, g)-alliance minimale en au plus $5n+4$ rondes additionnelles, où n est le nombre d'entités du système. $\mathcal{MA}(f, g)$ utilise $O(\log n)$ bits par entité et stabilise à une configuration terminale (légitime) en $O(\Delta^3 n)$ pas, où Δ est le degré de la topologie de réseau du système.

Travaux connexes

Le problème de (f, g)-alliance a été introduit par Dourado *et al.* [DPRS11]. Dans le même papier, les auteurs donnent plusieurs algorithmes répartis pour ce problème et ses variantes, mais aucun d'entre eux n'est auto-stabilisant. À notre connaissance, il s'agit de la seule publication sur les (f, g)-alliances jusqu'alors.

Cependant, il existe des résultats sur des instances particulières de (f, g)-alliances (minimales) [KM06, SX07, Tur07, WWTZ12]. Bien que ceux-ci considèrent aussi des réseaux quelconques identifiés, seule une solution est donnée avec la propriété de convergence sûre [KM06]. Srimani et Xu donnent un algorithme auto-stabilisant pour calculer une alliance défensive globale minimale en $O(n^3)$ pas [SX07], mais ils font cependant l'hypothèse que l'ordonnanceur est central. Turau donne un algorithme auto-stabilisant qui calcule un ensemble dominant minimal en $9n$ pas [Tur07], en présence d'un ordonnanceur (réparti) inéquitable. Wang *et al.* donnent un algorithme auto-stabilisant pour calculer un ensemble k-redondant dominant minimal en $O(n^2)$ pas [WWTZ12], en présence d'un ordonnanceur central. Enfin, un algorithme auto-stabilisant avec convergence sûre est donné par Kakugawa et Masuzawa pour calculer un ensemble dominant minimal [KM06]. L'algorithme calcule un ensemble dominant (pas nécessairement minimal) en $O(1)$ rondes et puis stabilise à un ensemble dominant minimal en $O(\mathcal{D})$ rondes, où $\mathcal{D}$ est le diamètre du réseau. Cependant, ils font l'hypothèse que l'ordonnanceur est synchrone.

Algorithme $\mathcal{MA}(f, g)$

Intuitivement, il faut que l'ensemble A compte suffisamment d'entités pour obtenir une (f, g)-alliance, et qu'il n'en compte pas plus que nécessaire pour qu'elle devienne minimale. En effet, si toutes les entités sont dans A, nous avons bien une (f, g)-alliance, mais elle est loin d'être minimale dans la plupart des cas. L'algorithme permet donc à toute entité de quitter ou de rejoindre A sous certaines conditions. Celles-ci sont nécessairement plus restrictives quand il s'agit de quitter A.

Quitter A

Pour obtenir la 1-minimalité, une entité p n'est autorisée à quitter A que si les deux prérequis suivants sont réunis :

1. p a suffisamment de voisins dans A, c'est-à-dire au moins $f(p)$, une fois qu'elle a quitté A.
2. Chaque voisin q de p a suffisamment de voisins dans A, c'est-à-dire au moins $g(q)$ ou $f(q)$ selon que q est dans A ou non, après que p a quitté A.

Pour assurer le premier prérequis, il suffit que p ait au moins $f(p)$ voisins dans A et qu'aucun de ses voisins ne soit autorisé à quitter A en même temps que p. Ainsi le retrait d'une entité de A est localement séquentiel.

Pour assurer le second prérequis, il suffit que tous les voisins de p aient suffisamment de voisins dans A (sans compter p) et qu'aucun voisin des voisins de p (sauf p) ne soit autorisé à quitter A en même temps que p.

L'algorithme évalue ces conditions par le biais d'un pointeur pour chaque entité autorisant son voisin ainsi pointé à quitter A. Pour pouvoir quitter A, une entité p doit avoir tous les pointeurs de ses voisins dirigés vers elle.

Chaque entité est dotée d'une variable booléenne de sortie indiquant à ses voisins si elle est dans A ou non. Cela permet à toute entité d'évaluer si elle a suffisamment de voisins dans A ou non. Si une entité présente un excédent de voisins dans A et qu'elle a plusieurs de ses voisins dans A, elle devrait simplement diriger son pointeur vers l'un d'entre eux.

Cette approche naïve pourrait mener à une situation d'interblocage si l'entité pointée ne peut pas quitter A, faute de remplir les deux prérequis énoncés ci-avant. Cette approche est donc complétée en dotant chaque entité d'une variable booléenne pour indiquer à ses voisins si elle est bloquée dans A, auquel cas ceux-ci ne dirigeront plus leur pointeur vers elle. Le positionnement de cette variable nécessite que chaque entité puisse évaluer elle-même si les deux prérequis sont remplis comme suit :

1. Le premier prérequis est aisément évaluable par une entité à partir du nombre de ses voisins dans A qui peut-être calculé en lisant leur variable de sortie.
2. Le second prérequis n'est pas directement évaluable par une entité, car il dépend du nombre de voisins dans A pour chacun de ses voisins. Un compteur est donc ajouté à chaque entité de façon à exposer ce nombre à ses voisins.

Une autre situation d'interblocage pourrait survenir d'une dépendance circulaire entre plusieurs entités via leurs pointeurs. Pour empêcher ceci de se produire, l'algorithme prévoit de diriger les pointeurs préférentiellement vers les entités ayant la plus petite identité, de façon à briser les éventuelles symétries.

Une dernière situation, de concurrence cette fois, peut se produire lorsqu'une entité a exactement un voisin en excédent dans A. En effet, si une telle entité change de voisin en même temps que le voisin précédemment pointé quitte A, puis que le voisin nouvellement pointé quitte aussi A, alors A n'est plus une (f, g)-alliance, car il manque un voisin dans A à cette entité. Pour éviter cela, l'algorithme impose que tout changement de pointeur vers un nouveau voisin doit d'abord passer par une réinitialisation à une valeur intermédiaire n'autorisant aucun voisin à quitter A, avant d'en pointer éventuellement un nouveau.

Rejoindre A

Toute entité p doit rejoindre l'ensemble A dans deux cas :

1. p a moins de $f(p)$ voisins dans A, ou
2. un voisin de p n'a pas assez de voisins dans A.

Ces conditions peuvent être localement évaluées par p en utilisant les mêmes variables que précédemment. De plus, pour empêcher p de quitter et rejoindre A en boucle, l'algorithme requiert que p ne soit pointé par aucun voisin pour pouvoir rejoindre A.

Partie V : Indexation dans les arbres ordonnés

Dans cette cinquième partie, nous abordons enfin le *problème d'indexation*. Pour chaque entité d'un système réparti, étant donnée une valeur d'entrée d'un type quelconque ordonné, appelée *poids*, il s'agit de trouver le rang de cette entité dans l'ensemble des entités ordonné par poids. Par exemple, l'entité de poids le plus faible doit être de rang 1, celle second plus petit poids de rang 2 et ainsi de suite.

Nous proposons ici un algorithme auto-stabilisant, appelé $\mathcal{RANK}$, qui indexe de la sorte les entités dans les systèmes répartis dont la topologie de réseau est un arbre ordonné. Cette solution stabilise en $O(n)$ rondes et requiert $O(\delta(p) \log n)$ bits par entité p, où n est le nombre d'entités dans le système et $\delta(p)$ est le nombre de voisins de l'entité p.

Il faut noter que cet algorithme n'est pas silencieux. Les rangs ne changent pas une fois que le système a stabilisé. Cependant l'algorithme recalcule ces rangs sans cesse. Si les poids ne changent pas, les calculs incessants de $\mathcal{RANK}$ seront transparents pour toute application qui utilise sa sortie.

Travaux connexes. Le seul algorithme auto-stabilisant précédent pour le problème d'indexation a été proposé par Bourgon *et al.* [BDN95]. Leur algorithme fonctionne dans les arbres enracinés. Comme le nôtre, il n'est pas silencieux. Il suppose que chaque entité a un identifiant unique dans l'intervalle $[1..n]$. Enfin, leur algorithme stabilise en $O(nh)$ rondes et nécessite $O(\log n)$ bits par entité, où h est la hauteur de l'arbre.

Par ailleurs, le problème d'indexation est lié au problème de tri où chaque entité, étant donnée une valeur d'entrée, doit maintenir une valeur finale de sorte que l'ensemble des valeurs finales soit l'ensemble des valeurs initiales, trié par-dessus le réseau. Il existe de nombreuses solutions auto-stabilisantes pour trier dans un arbre, par exemple [HP01, HM01, BDV05].

Algorithme $\mathcal{RANK}$. Notre algorithme s'appuie sur les paires de guidage calculées par l'algorithme $\mathcal{GUIDE}$ présenté dans la partie II. C'est même une composition collatérale hiérarchique (voir la partie I) de deux algorithmes : $\mathcal{RANK} = \mathcal{CRK} \circ \mathcal{GUIDE}$. Nous supposons que l'ordonnanceur est faiblement équitable pour les deux algorithmes. Nous supposons aussi que, pour chaque entité du système, pour chacun de ses voisins, cette entité peut déterminer si elle est ou non parente de ce voisin dans la topologie de réseau en arbre du système. L'algorithme $\mathcal{CRK}$, présenté ci-après, doit à la fois calculer correctement le rang de chaque entité du système, en utilisant les poids et les paires de guidage, et détecter les éventuelles erreurs afin de les corriger.

Calculer le rang. Le calcul global du rang est effectué par un cycle de quatre vagues.

- La première vague est lancée par l'entité à la racine de l'arbre ordonné et se propage en descendant dans l'arbre jusqu'aux feuilles. Il s'agit ici d'initialiser les variables internes au calcul du rang, notamment un compteur initialisé à zéro pour l'entité à la racine.

- La deuxième vague remonte dans l'arbre ordonné des feuilles vers la racine. Chaque entité située à une feuille de l'arbre ordonné crée un paquet ascendant composé de sa paire de guidage et de son poids. Chaque entité parente récupère de ses enfants les paquets ascendants qui ont un poids inférieur au sien, sinon elle crée son propre paquet ascendant. Les paquets sont ainsi acheminés, de parent en parent, jusqu'à la racine de l'arbre ordonné, un à un et par ordre de poids.

- La troisième vague redescend dans l'arbre ordonné. Pour chaque paquet ainsi récupéré, l'entité racine accroît de un son compteur pour attribuer le rang correspondant au poids. Elle crée alors un paquet descendant composé de la paire de guidage du paquet montant et de ce rang. Ce paquet est ensuite acheminé dans l'autre sens, jusqu'à l'entité correspondante qui récupère ainsi son rang, grâce aux paires de guidage.

- La quatrième et dernière vague remonte à nouveau dans l'arbre ordonné. Une fois que toutes les entités ont accusé réception de leur rang, l'entité à la racine de l'arbre ordonné en déduit que le calcul du rang est globalement terminé pour toutes les entités du système.

Corriger les erreurs. Il n'est pas possible de déterminer localement si un rang est erroné. Son calcul global est donc répété de façon cyclique afin de corriger les erreurs de rang pouvant être causées par des fautes transitoires. De plus, les entités vérifient à chaque pas la cohérence locale du calcul en cours. Quand une erreur est détectée par une entité, une alerte est propagée de proche en proche à toutes les entités potentiellement impactées. Cette alerte provoque chez chaque entité concernée la réinitialisation des variables internes consacrées au calcul du rang. Ceci permet d'éviter d'effectuer un cycle de calcul avec des données erronées et d'empêcher qu'une erreur causée par une faute transitoire puisse éventuellement empêcher de mener le calcul jusqu'à son terme.

Conclusion

Dans cette thèse, nous avons étudié la propriété d'auto-stabilisation appliquée à la construction de structures de données réparties.

Nous avons tout d'abord introduit les motivations de ce travail en présentant le domaine des systèmes répartis et plus particulièrement l'état de la recherche concernant l'auto-stabilisation. Nous avons défini les outils théoriques utilisés pour modéliser les systèmes répartis et qui servent de base aux raisonnements développés dans les parties suivantes de cette thèse.

Puis, dans la partie II, nous avons présenté d'une part les notions d'arbre couvrant à ensemble indépendant maximal (EIM), et d'autre part les paires de guidage, qui se sont montrées utiles à la construction d'autres structures. Nous avons d'abord donné un algorithme auto-stabilisant et silencieux qui trouve un arbre couvrant EIM dans une topologie quelconque en présence d'un ordonnanceur faiblement équitable. Après avoir prouvé la correction de cet algorithme et que son temps de convergence est linéaire en nombre de rondes, nous avons également montré que le problème qu'il résout est $\mathcal{P}$-complet. Ce dernier résultat nous a dissuadés de chercher un algorithme construisant un arbre couvrant EIM avec un meilleur temps de convergence. Cette structure est utilisée dans le k-partitionnement compétitif présenté dans la partie III. Ensuite, nous avons détaillé la notion de paires de guidage qui forment un étiquetage particulier des topologies en arbre. Ces paires de guidage sont notamment utilisées pour résoudre le problème d'indexation dans la partie V. Nous avons décrit un algorithme auto-stabilisant et silencieux qui calcule des paires de guidage dans une topologie en arbre quelconque et prouvé sa correction en présence d'un ordonnanceur faiblement équitable. Nous avons aussi montré qu'il converge en un nombre linéaire de rondes par rapport à la hauteur de la topologie en arbre.

Ensuite nous avons étudié le problème de k-partitionnement dans la partie III. Nous avons introduit les notions de k-partitionnement et d'ensemble k-dominant, puis montré une application possible du k-partitionnement et proposé deux approches allant vers l'optimisation. Nous avons corrigé une preuve qui établit une borne supérieure sur la taille d'un ensemble k-dominant minimum par rapport à la taille de la topologie. Nous avons proposé un algorithme auto-stabilisant et silencieux, inspiré par ce schéma de preuve, qui trouve un ensemble k-dominant minimal de taille bornée dans une topologie quelconque. Après avoir prouvé sa correction, nous avons montré qu'il converge en un nombre linéaire de rondes, en présence d'un ordonnanceur faiblement équitable. De plus, nous avons proposé un autre algorithme auto-stabilisant et silencieux qui calcule un k-partitionnement d'une topologie en arbre, en présence d'un ordonnanceur faiblement équitable. Nous avons prouvé qu'il trouve un k-partitionnement minimum dans une topologie en arbre quelconque. En le composant avec la construction d'arbre couvrant EIM que nous avons donnée précédemment, nous obtenons une solution plus générale pour les topologies quelconques. Nous établissons sa correction et son temps de stabilisation qui est linéaire en nombre de rondes. De plus, nous montrons que notre k-partitionnement est compétitif quand la topologie de communication du système réparti considéré est un graphe de disques unitaires (UDG) ou un graphe de disques unitaires approchés (ADG) qui est une généralisation du précédent. De tels graphes sont couramment utilisés pour modéliser des réseaux *ad hoc* sans fil. Enfin, nous avons évalué nos algorithmes par des simulations, présenté et analysé leurs résultats pour en déduire que le k-partitionnement compétitif aboutit aux meilleurs résultats en moyenne.

De plus, nous avons étudié le problème de construction d'une (f, g)-alliance minimale pour une topologie quelconque dans la partie IV. Il s'agit d'une généralisation de nombreux problèmes de construction de structure couvrante qui présentent un certain intérêt pour le domaine des systèmes répartis. Nous avons proposé une solution auto-stabilisante avec convergence sûre en présence d'un ordonnanceur inéquitable si $f \geq g$. Au-delà de la preuve de correction, nous avons montré que son premier temps de convergence est au plus quatre rondes tandis que son second temps de convergence est linéaire en nombre de rondes par rapport à la taille de la topologie.

Enfin, nous avons abordé le problème d'indexation pour les topologies en arbre dans la partie V. Nous avons proposé une solution auto-stabilisante qui converge en un temps linéaire en rondes en présence d'un ordonnanceur faiblement équitable et prouvé sa correction. Cet exemple d'application des paires de guidage démontre leur utilité pour la navigation dans les topologies en arbre.

Perspectives

Paires de guidage. Notre travail sur le calcul auto-stabilisant des paires de guidage peut être poursuivi selon plusieurs directions. D'abord, une extension directe de notre travail dans la partie II serait de prouver que notre solution fonctionne également en présence d'un ordonnanceur inéquitable, c'est-à-dire, qu'il ne nécessite

pas l'hypothèse d'un ordonnanceur faiblement inéquitable. Notre transformateur d'algorithme pour l'équité donné dans la partie I prouve qu'il existe une solution en présence d'un ordonnanceur inéquitable. Cependant, l'algorithme transformé stabilise en $O(\mathcal{D}n^3)$ pas, alors que nous conjecturons que notre solution actuelle stabilise en $O(nh)$ pas en présence d'un ordonnanceur inéquitable. Ensuite, davantage d'applications des paires de guidage devraient être étudiées, comme nous l'avons fait avec l'algorithme d'indexation réparti dans la partie V. Par exemple, nous avons pensé utiliser cet étiquetage pour implémenter un schéma de routage basé sur un partitionnement. En fait, une grappe est un ensemble connexe d'entités parmi lesquelles une entité est distinguée et appelée tête de grappe. Il est donc possible de construire un arbre couvrant d'une grappe enraciné à sa tête. Il se trouve qu'un tel arbre est déjà construit par notre algorithme de k-partitionnement décrit dans la partie III. Pour parachever la communication inter-grappes, les entités doivent être capables de communiquer dans les deux directions suivantes. Toute entité doit pouvoir envoyer un message à sa tête de grappe. Ceci peut être mené à bien en suivant simplement les liens parents de l'arbre couvrant sa propre grappe. Cependant, l'autre direction, c'est-à-dire de la tête de grappe vers un membre de sa grappe, peut s'avérer également très utile. Elle peut être efficacement prise en charge grâce aux paires de guidage.

k-Partitionnement. Ce qui nous amène aux possibles extensions de notre travail sur la construction auto-stabilisante de k-partitionnement présentée dans la partie III. D'abord, nous pensons qu'il est encore possible d'améliorer le temps de stabilisation de notre algorithme qui est actuellement en $O(n)$ rondes. Idéalement, nous aimerions proposer une solution auto-stabilisante qui converge vers un k-partitionnement en $O(k)$ rondes. Il faut remarquer que nous avons montré dans la partie II que notre construction d'arbre couvrant EIM ne pourrait pas permettre d'arriver à un temps si court, il faudra donc chercher une autre approche. Par ailleurs, nous aimerions étendre la construction de k-partitionnement *compétitif* à une classe de topologies de réseau plus générale que les UDG et ADG. Ici encore, une approche alternative à notre construction utilisant un arbre couvrant EIM doit être cherchée, car elle s'appuie fortement sur les propriétés de ces graphes pour obtenir cette compétitivité. Enfin, maintenant que nous avons prouvé leur correction, analysé leur complexité et simulé leur fonctionnement, nous aimerions faire un pas de plus dans l'étude de nos algorithmes en les déployant sur de vrais réseaux de capteurs sans fil. Ceci devrait permettre d'affronter de possibles problèmes d'implémentation, de mesurer leur réelle efficacité et d'étudier leur impact sur la consommation d'énergie.

(f, g)-Alliance. Dans la partie IV, notre solution auto-stabilisante avec convergence sûre au problème de construction d'une (f, g)-alliance minimale dans le cas où $f \geq g$ ouvre de nombreuses autres questions. La notion de (f, g)-alliance généralise un certain nombre d'autres structures couvrantes. Nous nous attendons à ce qu'il y ait, sur l'étude de la construction d'une (f, g)-alliance, de nombreuses implications de

résultats connus pour des structures particulières ; et réciproquement. Par exemple, les résultats d'impossibilité et les bornes de complexité sur la construction d'un ensemble dominant pourraient s'appliquer à celle d'une (f, g)-alliance. Cela pourrait également nous aider à améliorer la complexité en temps de notre solution, sans pour autant compromettre la complexité en espace. Enfin, il y deux autres cas que nous n'avons pas étudiés dans notre travail. Est-ce possible de construire efficacement une (f, g)-alliance dans le cas où $f < g$? La même question vient à propos du cas où f et g ne satisfont pas nécessairement la même inégalité pour toutes les entités du réseau.

D'autres horizons. Plus généralement, il serait intéressant d'étudier d'autres propriétés dérivées de l'auto-stabilisation. En particulier, la construction auto-stabilisante de structures couvrantes avec contention de fautes semble très prometteuse, car elle vise à confiner les fautes dans une petite partie du réseau. Cette partie pourrait idéalement correspondre à une subdivision de la structure couvrante en construction. Notons que l'intégralité de notre travail est écrit dans le modèle à mémoires localement partagées. Une extension directe de notre travail pourrait être de proposer des solutions auto-stabilisantes *efficaces* pour les mêmes problèmes étudiés dans le modèle à passage de message. Ce dernier modèle a l'avantage d'être proche de l'implémentation, donc l'efficacité des solutions proposées devrait être le principal but. Rappelons qu'il existe déjà des constructions générales pour ce modèle mais coûteuses. Pour finir, il serait intéressant d'aller au delà de l'étude des topologies de réseau non orientées, c'est-à-dire, de considérer les structures couvrantes comparables dans les topologies de réseau orientées quand cela a du sens.

Contents

List of Figures

List of Algorithms

List of Tables

List of Notations

Introduction

"Three little pigs live in a valley, where a wolf sometimes threatens houses with an enhanced battery-powered leaf-blower.

The house of Weak Pig was made of bricks, with a nice chimney and large windows, so he could enjoy the last warm rays of sunshine before sunset. One day, the wolf showed up in front of Weak Pig's house. The little pig quickly closed the windows. But the wolf climbed to the top of the roof and blew right down the chimney with his strong leaf-blower. The house ballooned so much that it burst, scattering the bricks far away, and the little pig was defenseless.

Pursued by the wolf, Weak Pig ran towards the house of Robust Pig. The house of Robust Pig was made of solid titanium, without a chimney or any windows. Actually, the only opening was the front door which could only be locked from the inside. Weak Pig jumped into the house and quickly locked the door. The two little pigs were safe from attack by the wolf's leaf-blower, as there was no way to reach into this house. However, due to the high cost of titanium, the house was very small, actually too small for both of them. They could not stay comfortably in it and decided to abandon the house.

Still pursued by the wolf, the two little pigs headed towards the house of Self-Stabilizing Pig. His house was made of latex in a fancy shape, which protected them. When the wolf's leaf-blower started up, the shape of the house changed, depending on the direction of the air current. In the meantime, the little pigs could not possibly use anything inside the house which included household appliances. Fortunately, the wolf could not break any part of the house, and so he eventually gave up his attack. Once the wolf stopped his leaf-blower, which incidentally ran out of battery, the house slowly went back to its initial shape again, without the three little pigs doing anything. The wolf was forced to step back for a while in order to return to his woods and recharge the battery of his high-powered machine.

Since then, the wolf has continued to try to catch the three little pigs, but not too often, as the battery of his house-blowing machine required a week charging time. In the meantime, when the wolf is away in the woods, the three little pigs are able to Self-Stabilizing Pig's household appliances safely."

The above tale, freely inspired by [HP86], illustrates the two following approaches to *fault-tolerance*. First, a system can be conceived to overcome every fault, when it is possible. It is often done at high cost or with drastically diminished functionalities as a whole. Otherwise, it can be conceived to recover from transient faults without external intervention. Then, the main drawback is the lack of reliability of functionalities while faults hit the system and during a finite period of time after faults cease to occur.

This latter approach is known as *self-stabilization*, which has been first proposed in [Dij74]. It assumes that faults are transient, that is, faults only hit the system in a rare and relatively short period of time. However, it does not make any assumption on the seriousness and the number of faults. Actually, self-stabilization even guarantees that the system can recover from any state.

This paradigm has been developed in the context of *distributed systems*. A system is said to be distributed when it is composed of several entities. Each entity executes its own program independently (without relying on any central control) and can communicate with an arbitrary subset of other entities (depending on the topology of the distributed system). However, a whole system still appears as a single system to its users. The gathering of every local program is called a *distributed algorithm*. It is the recipe that will enable a distributed system to provide the desired service to the user.

In order to overcome the apparent chaos that prevails in a distributed system, distributed applications have to achieve some type of coordination between entities. This can be achieved by the use of intermediate *distributed spanning structures*, that is, data structures which distribute the collected information over the whole system. The most famous of these structures in the literature of self-stabilization are the spanning tree and the clustering. The main purpose of the former is to avoid infinite loop of communication between entities. The latter organizes entities hierarchically, in order to implement higher-level applications such as routing, grid computing, distributed database, ... This type of structure is particularly useful to large-scale networks. The more entities there are in a distributed system, the more likely at least an entity of the system will be affected by faults. For instance, the Internet counts several billions of devices, including a lot of cheap computers which are prone to defects, so it is definitely going to suffer faults regularly.

This thesis more particularly focuses on the self-stabilization of distributed algorithms that construct distributed spanning structures over a distributed system. Here are the three main problems studied in this thesis:

- the construction of k-clustering which is a structure that organizes the distributed system hierarchically,
- the construction of (f, g)-alliance which is a structure that generalizes many types of alliance (such as the dominating set), and
- the ranking problem which consists in giving a rank to each entity of the distributed system, according to a distributed set of input values of ordered type.

For each of these problems, attention is paid to both the correctness and the efficiency of the solutions which are proposed. In the case of k-clustering, the various solutions of this thesis are also evaluated by running simulations.

Roadmap

The *first part* of this report sets down the general background of this thesis. In Chapter 1, the concept of self-stabilization is informally presented by giving an overview of related work. Like so, it explains the motivations that lie behind the study of self-stabilization, which is a whole subarea of fault-tolerant distributed algorithmic. The theoretical setting of this thesis is given in the two other chapters of this part as follows. Chapter 2 recalls some elements of graph theory used for reasoning on the topology of distributed systems. It also defines a few elements of automata, languages, and sets theories which are used for reasoning on the topology and the behavior of distributed systems. In Chapter 3, a model of computation for distributed algorithms is presented. The notion of distributed system, its execution driven by a daemon, and both self-stabilization and safe convergence are defined; the two latter being properties on distributed algorithms for fault-tolerance. A composition technique for self-stabilizing algorithms is also introduced in order to make both the writing and the understanding of the findings easier. Note that, for every distributed spanning structure studied throughout this thesis, this model is used to propose self-stabilizing constructions, prove their correctness, and analyze their time and space complexities.

In the *second part*, two distributed spanning structures are presented and are mainly used to support the construction of two other structures in the following parts of this thesis. In Chapter 4 first, the maximal independent set (MIS) tree is studied. It is a spanning tree such that the processes at even level form an MIS. Then in Chapter 5, a special labeling of processes in tree networks, called *guide pairs*, is studied. It provides a useful support for information navigation in trees.

In the *third part*, the problem of constructing a k-clustering in a self-stabilizing manner is considered. In Chapter 6, the notions of k-dominating set and k-clustering are defined and associated. Then, two different approaches are investigated to solve this problem. On the one hand, in Chapter 7, an upper bound on the size of minimum k-dominating set is studied and a self-stabilizing construction of small minimal k-dominating set based on this bound is proposed. This work has been essentially presented in a conference [DDH+11b] and more thoroughly detailed in a journal [DDH+13]. On the other hand, in Chapter 8, a self-stabilizing construction of a k-clustering is proposed. It is shown that, using the previously mentioned MIS tree construction, the resulting k-clustering is competitive in case unit-disk graphs (UDGs) or approximate disk graphs (ADGs) are used to model the network, typically a wireless sensor network (WSN). It is also shown that this solution computes a k-clustering with the minimum number of clusters in the case of tree networks. Part of this work has been already presented in a conference [DDH+12]. Finally in Chapter 9, both approaches are compared through experimentation.

In the *fourth part* (that is, in Chapter 10), the problem of constructing an (f, g)-alliance is described. A self-stabilizing solution with safe convergence is also provided. This strengthened variation of self-stabilization guarantees that, the system will quickly provide a minimum guaranteed service again, after being hit by transient faults, while continuously recovering from faults. This work has been accepted for presentation in a conference [CDD+13].

Next, in the *fifth part* (that is, in Chapter 11), the ranking problem is presented and a self-stabilizing solution which uses the aforementioned guide pairs labeling is given. This work has been briefly presented in a conference [DDLR11] and has been accepted for publication in a journal [DDLR13].

Finally, a report on the achievements of this thesis and the perspectives of future works is drawn up.

Hereafter, Figure 1 represents every part and every chapter of this thesis.

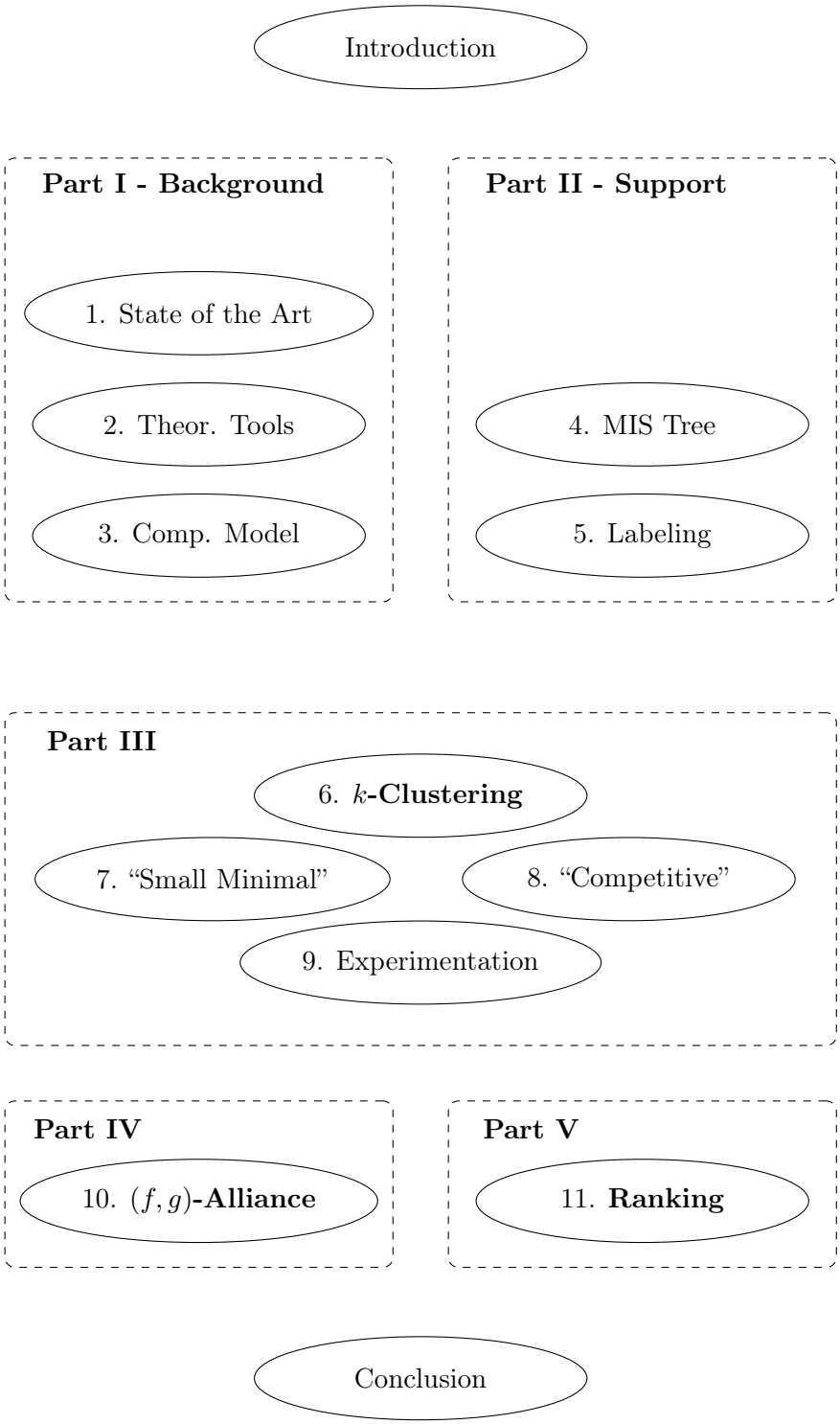


Figure 1 – Organization of this thesis.

Part I

Background

Chapter 1

State of the Art

Contents

In this chapter, a state of the art of self-stabilization is introduced. First, the general field of distributed systems is presented in Section 1.1 and some of its fundamental problems are presented in Section 1.2. In Section 1.3, the particular motivations behind the study of distributed structures in this thesis are explained. Afterwards, some aspects of dependability are presented in Section 1.4 and, more specifically, a way to improve it using fault-tolerance is presented in Section 1.5. Next, self-stabilization and its derived properties are discussed in Sections 1.6 and 1.7, respectively. Finally in Section 1.8, the different ways of modeling a distributed system in order to reason on self-stabilization are discussed.

1.1 Distributed Systems

In early ages, every computation was performed by hand, sometimes with the help of an abacus or a slide rule. Complex computations would be delegated to someone else whose profession was called "computer". It was the premise of distributed computing.

In 1946, the first electronic general-purpose computing machine was built and named ENIAC [Bur02]. It was the first computer designed to implement the universal Turing machine. Soon after, came the need for computers ability to communicate with each other. In the 1960s, computer networks turned a corner with the introduction of packet switching, first proposed in [Bar64], drastically lowering the cost of electronic data communications. During the same period, the computation power of computers was improved through the use of multiprocessing and multiprogramming, as summarized in [Cri63], resulting in cheaper computers. The 1980s saw the emergence of the Internet, the most famous computer network on Earth, which nowadays

connects billions of devices. Still, there are more current and further networks, such as wireless sensor networks (WSNs), peer-to-peer (P2P) architecture, grid computing, and mobile ad hoc networks (MANETs).

A *distributed system* is an inter-connected collection of independent entities, which appears as a single system to its users. As such, it is an abstract view of many distributed computing systems. Let us explain each term of this definition in the following paragraphs.

First, an *entity* here represents a computational unit of some type as follows: a computer, a core of a micro-processor, a process in the context of multitasking operating systems, ... The *independence* of entities is threefold: they are asynchronous, autonomous, and unaware of global details. Indeed, entities may not be equipped with a synchronized clock, and each entity may be influenced by its own environment. Thus, each entity may perform computations and communications at its own speed, and in this case that entities are said to be*asynchronous*. Entities are said to be *autonomous* in the sense that each of them is fitted with its own program and does not rely on any central control system. The local program at each entity usually does not include any global details of any kind, particularly on the distributed system itself. Global parameters, such as the number of entities in the system, are prone to change from a deployment to another, or even over time. We say that entities are *unaware of global details*, that is each entity executes its own program according to its sole local knowledge.

Moreover, these entities can communicate together, *connected* by various media, such as: twisted pair wires, high-frequency radio-waves, ... Note that this does not mean that every entity is directly connected with each other. Still each entity can indirectly broadcast information to the whole distributed system through gossiping, in the literal sense.

Finally, users of such systems do not need to know their internal architecture in order to benefit from their service. This allows a great modularity in the implementation of a service. For example, we do not care if the postal services gather everything into a central sorting office or if there are some decentralized depots to handle the letters we send and receive. Similarly, most users of the World Wide Web do not care to know how the webpage they requested has been built and delivered to them. There are many other examples of this, involving computers or not. So, the use of distributed systems has really been widespread for a long time.

1.2 Fundamental Distributed Problems

A *distributed algorithm* is designed to run on a given distributed system in order to achieve a global task, whose inputs and outputs are scattered among entities.

Here, we distinguish parallel algorithms from distributed algorithms. Parallelization is a technique to perform a huge computation by cutting it into small chunks which can be computed independently on several entities. It allows to achieve such computation much faster. However, parallel algorithms usually run in a controlled

system, that is the computation is decentralized, but the control remains central.

Distributed algorithms go beyond the pooling of computing powers achieved by parallel algorithms in a controlled environment. They favor the independence between entities, in particular, entities are still assumed to be autonomous. It allows for more flexible implementation and easier deployment. However, they have to address some other types of problems. Some of them have been identified as fundamental in the building of most of higher-level applications in distributed systems. Some examples which highlight the main characteristics of distributed systems are given below.

Routing. It is often necessary to send information from one place of a distributed system to another or every other place. However, recall that not every pair of entities in a distributed system is actually directly connected. Instead they have to communicate indirectly through a chain of entities which will forward information between them. Usually, every entity keeps a routing table in the following form: for each destination, which entity information must be forwarded to? The *routing* problem is to find, for every entity, for every possible destination, which entity information has to be forwarded to next. This becomes even more difficult if the network is dynamic, that is some entities or connections can be either added or removed over time, such as the Internet. A survey on this topic can be found in [Raj02]. In Chapter 6, we refer to the routing as a major application of our contributions in Part III.

Resource Allocation. When an application requires a given number of resources for its own functioning, the following question arises: how to make sure that all these resources will be at the disposal of this application at the same time? This problem is mainly related to concurrence and conflicts resolution. Resources are usually attached to a distributed system in a limited set. Actually, they are available from some entities only, not from any place of the distributed system. The number of entities is very large compared to the number of resources. A resource can be used by at most one entity at a time. Moreover, they can even change their location in the distributed system. Because of the lack of global knowledge, an entity which needs to use some resources must search it through the distributed system. Typically resources are peripherals. An interesting special case of resource allocation is the *dining philosophers problem* briefly introduced in [Dij65] and further detailed in [Hoa85]. A more general case of this problem is proposed in [CM84].

Mutual Exclusion. This is a well-known resource allocation problem. There is only one resource and this can be only used by at most one entity at a time. In order to fulfil this requirement, entities have to collaborate, so as to know when they can use the resource and when they must not use it. Even in this rather simplified setting, this problem is not trivial, due to entities being asynchronous and having local-only knowledge of the distributed system. One of the founding articles on this topic is [Lam78].

Consensus. A distributed system can sometimes be required to have all its entities hold a single value for further applications. As each entity may propose a different value independently, they have to agree on a single value all together. In the *consensus problem*, each entity is given an initial Boolean value and all entities must agree on the same Boolean value with the following restrictions: the final value must be decided among the initial values, every entity can decide a value at most one time, and every entity must eventually decide the same value. Although the statement of this problem looks simple, it is actually impossible to solve in many cases, as shown in [FLM85] and [FLP85].

Leader Election. Recall that in a distributed system, without further hypothesis, there is no central control. For many applications, it is very convenient to have only one entity taking the decisions for all the others. The *leader election* is the problem of distinguishing a unique entity among all entities of the whole distributed system. This problem has been first introduced in [LL77, page 158]. Electing a leader among the entities allows to have one entity taking the decisions for the whole distributed system, that is, centralizing the control of the system. Note that it is also a means to achieve synchronization between all entities.

1.3 Distributed Structures

A *distributed structure* is a means to organize a distributed system. Such organizations are often used in higher-level applications. It is especially interesting in the case of large-scale networks. However, they are particularly tedious to construct, because they are global to the distributed system by nature, whereas entities only have a local knowledge. Three examples are given below, the distributed construction of which this thesis is about.

Spanning Tree. As seen in the aforementioned routing problem, it is often useful to have every pair of entities able to communicate indirectly through a chain of entities between them. Obviously, this chain has to be of finite length. However, this is not always the case, because of possible cycles in the chain, causing the communication to go through the same entity an infinity of times. For example, if there are at least three distinct entities between the two entities that want to communicate together.

Informally,[1] a *tree* is a subnetwork of a distributed system, such that for each pair of entities, there is an unique chain of distinct entities between them, that is without any cycle. When all entities are part of the tree, we say that this tree is *spanning* the distributed system.

[1] A formal definition of spanning tree is given in Chapter 2.

In many cases, and in this thesis in particular, every spanning tree is considered to distinguish one of the entities among the others. This unique entity with respect to the spanning tree is called the *root*. Because of its uniqueness, the root is also a leader of the distributed system, as defined in the leader election problem mentioned in the previous section. It does not come as a surprise that these two problems are tightly connected. In fact, most solutions to the leader election problem are given as part of the distributed construction of a spanning tree. A survey on this problem, in the context of self-stabilization, is given in [Gär03]. A specific instance of spanning tree is studied in Chapter 4.

Clustering. Another way to achieve a hierarchical organization of a distributed system is to partition it into distinct *clusters*, such that each cluster is an inter-connected subset of entities and one of these entities is distinguished and called *clusterhead*. In such a partition, called *clustering*, of a distributed system, every entity belongs to a cluster and obeys its clusterhead. Different types of distributed clustering are presented in [Bas99, BK01, FM02]. A specific type of clustering is studied in Part III. Formal definitions and advanced details on clustering are given in Chapter 6.

Alliance. Alliances of entities are meant to have a global control of a distributed system, solely using local knowledge of entities. The definition of alliance in graphs was first introduced in [KHH04]. An *alliance* is a nonempty subset of entities such that it ensures, for every entity, some property on the number of allied neighbors and the number of neighbors which are not in the alliance. An alliance is *defensive* if, for every ally, there is at least as many allied neighbors as neighbors which are not in the alliance. Its construction is a generalization of some distributed problems. For instance, a *dominating set* is a defensive alliance such that every entity which is not in this set has at least one dominating neighbor. An extensive early bibliography on this problem can be found in [HL90]. The notion of defensive alliance is also closely related to the one of web community in the Internet, as pointed out in [FR07], which is defined as a set of entities, such that every member has a majority of neighbors which are within its own community too. A specific case of alliance is studied in Part IV.

1.4 Dependability

From the very beginnings of computing machines, availability, reliability, and maintainability have been constant concerns. They are enclosed in the concept of *dependability* proposed by Jean-Claude Laprie in the 1980s, which is intuitively defined as "the ability to deliver service that can justifiably be trusted" [ALRL04]. It involves the following self-explanatory attributes: availability, reliability, safety, confidentiality, integrity, and maintainability.

Here, we are particularly interested in the *reliability* of distributed systems and the service they provide. For example, if a distributed system is requested to decide on an unique value from a set of values distributed among the system, that is solving the consensus problem, then we want this system to provide an unique value continuously at any time. Note that it does not necessarily have to be always the same value.

On the one hand, the dependability of a system can be lowered by threats on any of its attributes. When a system is turned on, it provides a service which complies a specification. However, an event, called *fault*, may cause the system to behave unexpectedly, that is, to enter an unexpected state. Faults will be characterized with more details in the next section. The difference between the actual state and the expected state of a system is called an *error*. Such an error may result in a *failure* of the system to follow its specification. Note that, a failure may trigger a fault in its turn, as illustrated in Figure 1.1.

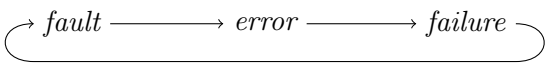


Figure 1.1 – Causality between faults, errors, and failures of a system.

On the other hand, the dependability of a system can be improved by the following approaches. In order to justify the trust in the ability of a system to deliver a service, we may want to simply *forecast* the number of faults that (will) affect the system and their incidence on its behavior. If this number is too high or cannot be evaluated, we can then try to *remove* as many faults as possible from the system. If we cannot remove all of them, we would like to *prevent* any of the remaining faults from occurring in the system. When all the aforementioned approaches are impracticable, we should design our algorithm so as to avoid failures, even in the presence of faults, that is, a *fault-tolerant* algorithm. This is particularly important to large-scale distributed systems, because they are particularly prone to hardware malfunctioning and they cannot afford to restart algorithms each time a fault occurs.

1.5 Fault-Tolerance

Here, faults which may occur in a system are being characterized in relation to time: duration of presence and frequency of occurrences. First, some faults can be *permanent*, such as the crash of an entity, or the presence of a malicious entity [LSP82]. On the contrary, the presence of *temporary* faults is bounded in time. When temporary faults repeatedly occur at regular period of time, they are said to be *intermittent*. For example, this is the case of an unreliable connection between two entities that would drop one message over two on the average. Finally, temporary faults which occur rarely are said to be *transient*, the casual loss of message from a healthy connection.

Now, two different strategies can be adopted to tolerate faults, depending on whether failures are unacceptable or not.

In the first one, algorithm is required to follow the specification absolutely without any failures, even temporary ones. This is called *robustness*. Originally, it was one of the motivations behind the switch from central systems to redundant architectures: the robustness of a single system can be improved by replicating it, such as, for example in the control system of the Space Shuttle Aircraft [GS84]. Robust algorithms must mask any possible error in the system state. This approach is *pessimistic* in the sense that algorithm has an extremely limited trust in the reliability of the underlying system. Such stringent avoidance of failures is not always possible [FLP85] to achieve and generally has a high cost in both time and space requirements.

In the other one, some failures can be accepted, as long as they are short and rare compared to the lifetime of the system and the availability requirements of the application. Then an *optimistic* approach is possible as follows. Errors are not necessarily masked to the application and may result in failures to conform the top-level specification, then the algorithm must handle the errors by itself to recover. However, errors are assumed to be caused by transient faults only. The only requirement here is that there must be fault-free periods of time which are not any much longer than the recovery time of the algorithm. This approach allows for efficient solutions in a hostile or unreliable large-scale distributed system such as the Internet. Self-stabilization, presented in the next section, deals with such optimistic, yet not error-masking, approach to fault-tolerance.

1.6 Self-Stabilization

Self-stabilization is a versatile technique to withstand any transient fault in a given distributed system. It was first introduced in [Dij74] and highlighted in [Lam85]. Informally,[2] a distributed algorithm is self-stabilizing if, after transient faults hit the system and place it in some arbitrary configuration, the system recovers without external (*e.g.*, human) intervention in finite time. Thus, self-stabilization makes no hypothesis on the nature or extent of transient faults (except that they could not hit the code) and recovers from the effects of those faults in a unified manner. To this purpose, self-stabilizing algorithms do not rely on the initial state of the system. This makes the deployment of a large number of entities easier, since their memory is not required to be specifically initialized. To a certain extent, distributed self-stabilizing algorithms are also prone to tolerate some *detectable* topological changes in the context of dynamic networks. In that case, topological changes have to be rare enough just as transient faults are. Some self-stabilizing algorithms for dynamic networks have been proposed in [DIM89].

[2] Self-stabilization is formally defined in Chapter 3.

However, self-stabilization has some drawbacks. First, the design of a self-stabilizing solution to a distributed problem generally induces an *overhead* compared to a non self-stabilizing one. This overhead can be in terms of either memory requirements, or execution time, or both of them. Next, self-stabilization consists in a global recovery of the distributed system. Thus, entities cannot locally detect if the system has already recovered. This makes the *termination detection* of self-stabilizing algorithms for distributed problems impossible. Finally, the main drawback is the *temporary loss of safety*. After the occurrence of transient faults, there is a finite period of time — called the *stabilization phase* — before the system returns to a legitimate configuration. During this phase, there is no guarantee of safety, in the sense that the system may not conform its specification during that time. Some attempts have been made to mitigate those drawbacks by proposing alternatives to self-stabilization which will be summarized in the next section.

1.7 Derived Properties

Several approaches have been introduced to offer more stringent guarantees during the stabilization phase.

Fault-Containment. When a small number of transient faults, compared to the number of entities, hit the system, the concept of *fault-containment* [GGHP96] allows to restrict the number of transitively affected entities. The idea is that transient faults that hit some entities should not be propagated through the whole distributed system so that it can recover quickly. Fault-containing algorithms are self-stabilizing and guarantee that, when few faults hit the system, those faults are confined within a preset radius around the affected entities and the stabilization time is short.

Superstabilization. *Superstabilization* was introduced in [DH97] in the view of dynamic networks. It enables self-stabilizing algorithms to additionally tolerate topological changes as follows. In the presence of single topological changes, which are assumed to be locally detected by affected entities, a superstabilizing algorithm quickly recovers and guarantees that a *passage* predicate is satisfied during that recovery. The complexity of superstabilizing algorithms is evaluated on both the maximum recovery time and the maximum number of entities that must change their state in order to handle a single topological change.

Time-Adaptivity. From the perspective of minimizing the stabilization time, the notion of *time-adaptivity* was first introduced in [KPS97] and refined in [BHKPS05]. It is almost the same concept as fault-locality proposed in [KP95b]. It only applies to non-reactive problems, which consist in computing an output from the input of the distributed system. Time-adaptive algorithms are self-stabilizing and guarantee a short recovery time of the output in the case of a small amount of faults. More precisely, if the system is in a legitimate configuration and is hit by transient faults that corrupt the state of

some f entities, then the output of the system self-stabilizes within $O(f)$ time. Note that, while the output self-stabilizes proportionally to f, the global state (which contains all the variables of the algorithm) of the system itself is not guaranteed to self-stabilize that fast. That is, its stabilization time depends on the diameter $\mathcal{D}$ or the size n of the system topology.

Safe Convergence. We consider the notion of *safe convergence* in Chapter 10 which was first introduced in [KM06]. The main idea[3] behind this concept is the following: for a large class of problems, it is often hard to design self-stabilizing algorithms that guarantee small stabilization time, even after few transient faults [GT02]. Large stabilization time is usually due to strong specifications that a legitimate configuration must satisfy. The goal of a *safely converging self-stabilizing algorithm* is to converge quickly ($O(1)$ rounds is usually expected) first to a *feasible* legitimate configuration, where a minimum quality of service is guaranteed. Once such a feasible legitimate configuration is reached, the system continues to converge to an *optimal* legitimate configuration, where more stringent conditions are required. Safe convergence is especially interesting for self-stabilizing algorithms that compute optimized data structures, *e.g.*, minimal dominating sets [KM06], approximation of the minimum weakly connected dominating set [KK07], and approximately minimum connected dominating set [KK12].

1.8 Models

The primary model in distributed algorithmic is called the *message-passing* model. It is closely inspired by real computer networks, but applies to any distributed system. Entities are connected by communication channels. At one end of the channel, the sender puts a message. The channel acts like a timed queue of messages. At the other end, the receiver queries the channel for a message and receives it the time transmission expired.

It is proven that solutions to a large class of distributed problems (namely the *suffix-closed* problems) in the message-passing model admit a self-stabilizing extension in the same model, according to the automatic transformation method given in [Kat93]. This transformer uses snapshots [CL85] which basically are records of the whole system configuration. Such technique induces a large overhead in terms of both time and space. So, this transformation method proves the existence of self-stabilizing solution to distributed problems that have a non self-stabilizing one; however, it is not expected to provide an *efficient* self-stabilizing solution. A more efficient transformation method has been previously proposed in [AV91], but it makes the strong assumption that entities are synchronous in the system.

At a higher level of abstraction, there is the so-called *locally shared memory* model. It is strongly inspired by multiprocessing computers. Entities are connected

[3] A formal definition of *safe convergence* is given in Chapter 3.

through shared communication registers. Each entity may atomically read in a set of registers and write in a possibly different set of registers. This model considerably makes the writing of distributed algorithms easier. In contrast to the previous model, they cannot be straightforwardly implemented on actual hardware, it requires some additional work. Most self-stabilizing algorithms in the literature have been written for the shared memory model, in the form of guarded actions. It is also the model we use in this thesis, it is formally defined in Chapter 3.

Because of the differences between these two models, the ability to switch from one model to another is an essential matter in the field of distributed algorithmic. Several methods have been proposed to rewrite non self-stabilizing algorithms from a model to another [AW91, BND89]. The most difficult part is to simulate the shared memory model using the message-passing model, which is more fine-grained. In particular, the boundedness of communication channels and entities memory can determine whether a self-stabilizing solution to a class of problems is possible as pointed out in [GM91]. The reliable message transmission problem and the propagation of information with feedback are the key elements of methods that preserve the self-stabilization property [GM91, AB93, Var94]. The first of these methods [GM91] relies on unbounded memory at each entity. Whereas the second proposed method [AB93] uses an infinite sequence of randomly generated numbers instead. And the third method [Var94] makes the assumption of bounded communication channels.

Note that most of our algorithms are silent. Now, most of such algorithms can be transformed into equivalent algorithms in the message-passing model using a simple (heartbeat-based) method [DDT05], which does not require any additional assumption on the system, and even tolerates intermittent loss of messages.

Finally, other models have been proposed to offer different trade-off between ease of writing algorithms and ease of actual implementation, *e.g.*, the finite-state message-passing model [HNM02].

Chapter 2

Theoretical Tools

Contents

In this chapter, we first present some elements of graph theory used for reasoning on the topology of distributed systems. Then, we recall few elements of automata, languages, and sets theories, used for reasoning on the behavior of distributed systems.

2.1 Elements of Graph Theory

2.1.1 Basics

A (simple) *graph* $G = (V, E)$ (actually a *directed* graph or *digraph*) is composed of a finite set V of *vertices* and a (finite) set E of *edges* – which are ordered pairs of distinct[1] vertices – such that $E \subseteq V^2$. We denote by n the number of vertices $|V|$.

A graph $G = (V, E)$ is *undirected* if and only if, for every edge (u, v) in E, there also exists an edge (v, u) in E. Equivalently, E is a set of unordered pairs and we denote $\{u, v\}$ both (u, v) and (v, u). Unless explicitly mentioned, we only consider here undirected graphs. So from now on, we omit to specify "undirected".

A *path* is a sequence $v_0, v_1, \ldots, v_\ell$ of vertices, such that, $\forall i \in [0..\ell-1], (v_i, v_{i+1}) \in E$. The *length* of a path $v_0, v_1, \ldots, v_\ell$ is the number ℓ of edges composing the path. A path $v_0, v_1, \ldots, v_\ell$ is *elementary* if and only if every vertex occurs at most once in the path, *i.e.*, $\forall i, j \in [0..\ell], v_i = v_j \Rightarrow i = j$.

[1] *Self-loops* are not allowed here.

A *cycle* is a path $v_0, v_1, \ldots, v_\ell$ where the starting vertex is also the ending one, *i.e.*, $v_0 = v_\ell$. A path $v_0, v_1, \ldots, v_\ell$ is an *elementary* cycle if and only if $v_0, v_1, \ldots, v_{\ell-1}$ is an elementary path and $v_0 = v_\ell$.

A graph $G = (V, E)$ is *connected* if and only if, for every pair u, v of vertices, there exists a path from u to v in G.

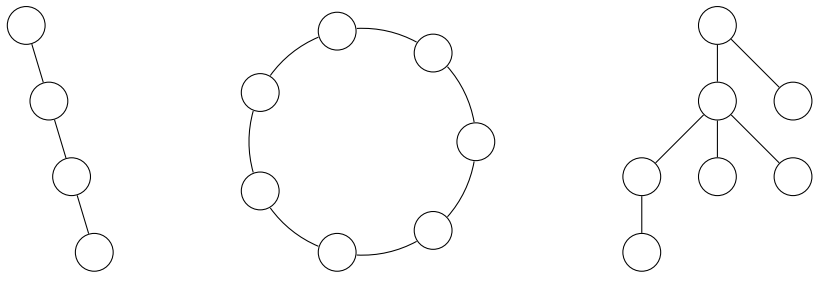
Given a connected graph $G = (V, E)$ and two vertices u and v in V, the *distance* from u to v is the length of the shortest path from u to v and is denoted by $\|u, v\|_G$ or simply $\|u, v\|$ when it is clear from the context.

The *diameter* of the connected graph G is the maximum distance between every two vertices of G. We denote the diameter of G by $\mathcal{D}(G)$ or simply $\mathcal{D}$ when it is clear from the context.

Given a graph $G = (V, E)$ and two vertices u and v in V, u and v are *neighbors* if and only if $\{u, v\} \in E$. The *degree* of a vertex v is the number of its neighbors and is denoted by $\delta_G(v)$ or simply $\delta(v)$ when it is clear from the context. The *average degree* of the graph G is the average of vertex degrees of G and is denoted by $\overline{\delta}(G)$ or simply $\overline{\delta}$ when it is clear from the context. The *degree* of the graph G is the maximum degree of every vertex and is denoted by $\Delta(G)$ or simply Δ when it is clear from the context.

A *subgraph* of a graph $G = (V, E)$ is any graph $G' = (V', E')$ such that $V' \subseteq V$ and $E' \subseteq E$. Given a subset V' of V, the subgraph of $G = (V, E)$ *induced* by V' is the graph $G' = (V', E')$ such that $E' = \{\{u, v\} : u, v \in V' \wedge \{u, v\} \in E)\}$.

A graph $G = (V, E)$ is *isomorphic* to another graph $G' = (V', E')$ if and only if there exists a function $f : V \to V'$ such that $\forall u, v \in V, \{u, v\} \in E \Leftrightarrow \{f(u), f(v)\} \in E'$. A *chain* is a connected graph which is isomorphic to an elementary path. A *ring* is a connected graph which is isomorphic to an elementary cycle. A *tree* is a connected graph containing $n - 1$ edges. In particular, a tree is *acyclic*. These three types of graph are represented in Figure 2.1.



(a) Chain (b) Ring (c) Tree

Figure 2.1 – Examples of graph.

2.1.2 Rooted Tree

A tree can be *rooted* at some vertex, meaning that one of its vertices, noted r, is distinguished as the *root*. Let $T = (V, E)$ be a tree rooted at vertex r. The *level* of a vertex v in T, is the distance of v to the root r, *i.e.*, $\|v, r\|_T$. We denote it by $lvl_T(v)$ or simply $lvl(v)$ when unambiguous. The *height* of the tree T, noted $h(T)$ or simply h when T is understood, is the maximum level for every vertex v of the tree, *i.e.*, $\max_{v \in V} lvl_T(v)$.

The *parent* of a vertex v in T, denoted by $par_T(v)$ or simply $par(v)$ when it is clear from the context, is v itself if $v = r$, otherwise it is its (unique) neighbor u such that u is in the shortest path from v to r, *i.e.*, $\|v, r\|_T = \|u, r\|_T + 1$. A *child* of v in T is any neighbor u such that $par_T(u) = v$. We denote by $chl_T(v)$ or simply $chl(v)$ when it is clear from the context, the set of all children of v in T. Two distinct processes u and v are said to be *siblings* in T if and only if they have the same parent, *i.e.*, $par(u) = par(v)$.

An *ancestor* of a vertex v in T is any vertex u in the shortest path from v to r. A *descendant* of a vertex v in T is any vertex w such that v is in the shortest path from w to r.

Given a rooted tree $T = (V, E)$ and a vertex $v \in V$, the *subtree* of T rooted at v, noted $T(v)$, is the subgraph induced by the descendants of v in T.

An *ordered tree* T is a rooted tree, together with a local *left-to-right order* on the children of each vertex v, denoted by $\prec_v$.

A *spanning tree* of a connected graph $G = (V, E)$ is any connected graph $T = (V_T, E_T)$ such that $V_T = V$, $E_T \subseteq E$ and $|E_T| = |V_T| - 1$. Any spanning tree becomes a rooted tree by choosing a distinguished root r; here, all spanning trees are rooted.

We define a *breadth-first search tree* (BFS tree) rooted at r, of a graph $G = (V, E)$ to be any spanning tree T rooted at r such that the path, through T, from any vertex v to r has length $\|v, r\|_G$ (the actual distance from v to r in the graph G). An example of BFS tree is given in Figure 2.2.

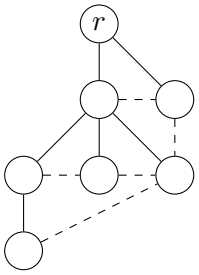


Figure 2.2 – Example of BFS tree rooted at some vertex r. Edges which are not part of the BFS tree are represented by dashed lines.

2.1.3 Independent Set and Dominating Set

An *independent set* of a graph $G = (V, E)$ is any subset I of V such that no two (distinct) members of I are neighbors in G, *i.e.*, $\forall u, v \in I, \|u, v\| \neq 1$. An independent set I of a graph $G = (V, E)$ is said to be *maximal* if and only if no proper superset of I is an independent set of G.

A *dominating set* of a graph $G = (V, E)$ is any subset D of V such that every vertex not in D has at least one neighbor in D, *i.e.*, $\forall v \in V : v \in D \vee (\exists u \in D : (u, v) \in E)$. A dominating set D of a graph $G = (V, E)$ is said to be *minimal* if and only if no proper subset of D is a dominating set of G.

Given $k > 0$, a *k-dominating set* of a graph $G = (V, E)$ is any subset $D(k)$ of V such that every vertex in V is at distance at most k of at least one vertex in $D(k)$, *i.e.*, $\forall v \in V : (\exists u \in D(k) : \|u, v\|_G \leq k)$. A k-dominating set $D(k)$ of a graph $G = (V, E)$ is said to be *minimal* if and only if no proper subset of $D(k)$ is a k-dominating set of G.

2.1.4 Unit-Disk Graph and Approximate Disk Graph

The topology of wireless sensor networks (WSNs) is constrained by the physical position of sensors and the range of transmission of their radios. Here we recall two models used to represent such networks.

Let $G = (V, E)$ be a graph where vertices are fixed points in the Euclidean plane $\mathbb{E}^2$. For any pair of vertices u and v, we denote by $\|u, v\|_{\mathbb{E}^2}$ the Euclidean distance between them in the plane $\mathbb{E}^2$.

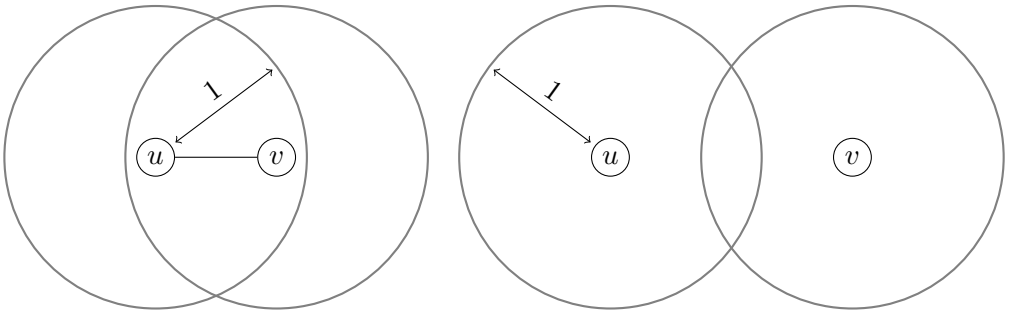
The graph G is a *unit-disk graph* (UDG) if and only if, for every pair of vertices in V, they are connected by an edge in E if and only if their Euclidean distance is at most one, *i.e.*, $\forall u, v \in V, \{u, v\} \in E \Leftrightarrow \|u, v\|_{\mathbb{E}^2} \leq 1$. Figure 2.3 illustrates the two possible cases for the connectivity of a pair of vertices in a UDG.

The graph G is an *approximate disk graph* (ADG) with *approximation ratio* λ if and only if, for every pair of vertices u and v in V:

1. $\|u, v\|_{\mathbb{E}^2} \leq 1 \Rightarrow \{u, v\} \in E$; and
2. $\{u, v\} \in E \Rightarrow \|u, v\|_{\mathbb{E}^2} \leq \lambda$.

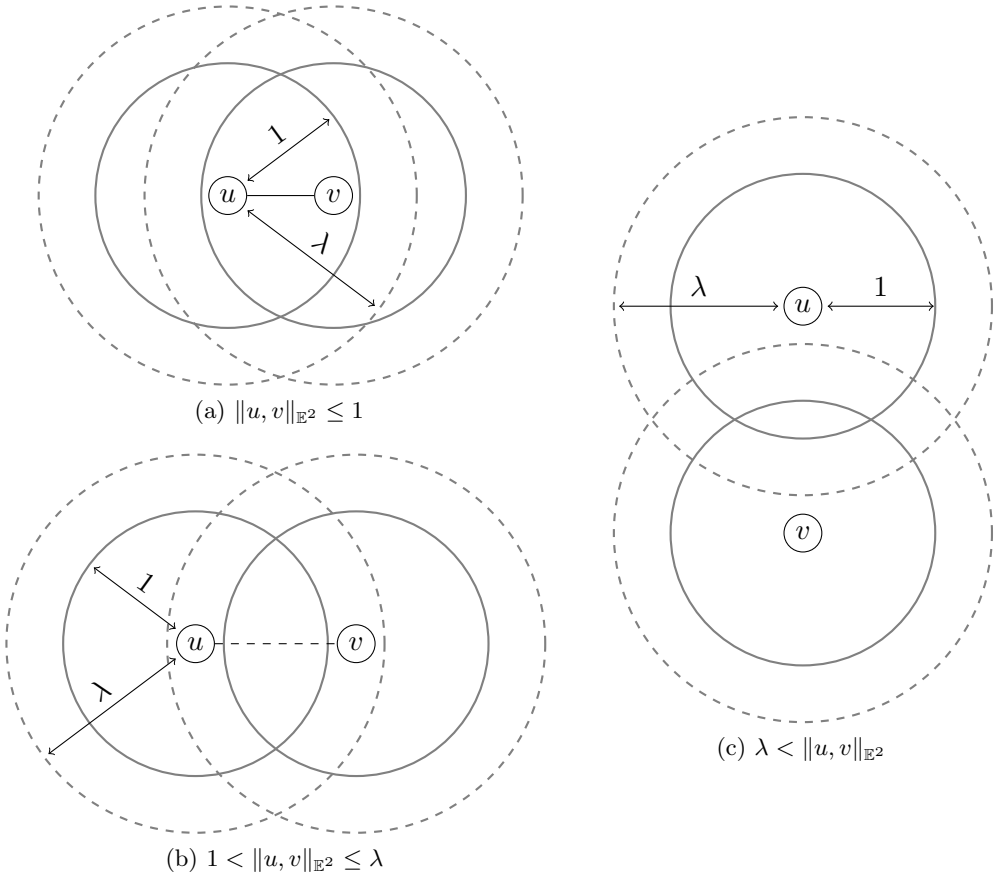
Figure 2.4 illustrates the three possible cases to decide whether or not two vertices of an ADG are connected.

In unit-disk graphs, we assume that the Euclidean distance is the only factor to determine whether two processes can communicate. This is not always the case in WSNs, because of radio-frequency interferences, since a sensor may steadily communicate with another sensor which is not his closest neighbor, as shown in [SMP99]. The class of ADGs has been first introduced by [BFN01], to circumvent these lacks. It is also known as *quasi unit-disk graph* (quasi-UDG), from [KWZ03].



(a) $\|u, v\|_{\mathbb{E}^2} \leq 1$ (b) $1 < \|u, v\|_{\mathbb{E}^2}$

Figure 2.3 – Examples of unit-disk graph (UDG). Vertices in (a) are connected, not in (b), according to their position in the Euclidean plane.



(a) $\|u, v\|_{\mathbb{E}^2} \leq 1$ (c) $\lambda < \|u, v\|_{\mathbb{E}^2}$

(b) $1 < \|u, v\|_{\mathbb{E}^2} \leq \lambda$

Figure 2.4 – Examples of approximate disk graph (ADG) with approximation ratio λ. Vertices in (a) are connected, vertices in (b) may be connected, vertices in (c) are not connected, according to their position in the Euclidean plane.

2.2 Elements of Automaton Theory

Transition System. A *transition system* is made up of

- a set $\mathcal{S}$ of states,
- a binary transition relation $\mapsto$ between states of $\mathcal{S}$, *i.e.*, $\mapsto \subseteq \mathcal{S} \times \mathcal{S}$, and
- a set $\mathcal{I}$ of initial states, such that $\mathcal{I} \subseteq \mathcal{S}$.

We denote a transition system by $(\mathcal{S}, \mapsto, \mathcal{I})$.

If we are not interested in restricting the set of initial sets, which is always the case when modeling self-stabilizing algorithms, we consider $\mathcal{I} = \mathcal{S}$ and simply denote such a transition system by $(\mathcal{S}, \mapsto)$.

2.3 Elements of Language Theory

Let S be a set, then:

- $S^+ = \bigcup_{n \in \mathbb{N}^*} S^n$ is the set of all finite sequences over S;
- $S^\omega = S^{\mathbb{N}}$ is the set of all infinite sequences over S;

2.4 Elements of Set Theory

Let S be a set and S' be a subset of S, then:

- $\mathbb{1}_{S'} : S \to \{0, 1\}$ is a function that outputs 1 if and only if the input is a member of S'.
- $\mathcal{P}(S) = \{S'' \subseteq S\}$ is the set of all subsets of S.

Chapter 3

Computational Model

Contents

A *distributed system* is a set of n communicating *processes*. In this chapter, we first define the processes and their communication capabilities. Next, we define a distributed algorithm according to the *locally shared memory model*. We then define self-stabilization and some of its related properties, and propose a composition technique for distributed algorithms. Finally, we give a general method to efficiently transform a self-stabilizing weakly fair algorithm into a self-stabilizing algorithm working under an unfair daemon.

3.1 Process

In this thesis, a *process* is an autonomous computational unit, which can be modeled by a deterministic automaton. Each process has a *unique identifier* (UID). The set of UIDs is totally ordered by $\prec$. Each UID is stored on b bits. For the purpose of complexity analysis, since we need to represent n different UIDs, we make the usual assumption that $b = O(\log n)$. We denote the UID of a process p by p.

Unless explicitly mentioned, processes have no a priori global knowledge about the distributed system. In particular, they do not know of n, other UIDs, *etc.*

3.2 Communication and Topology

Each process is able to *communicate* with a subset of other processes. Two distinct processes which can communicate together are said to be *neighbors*. The communication capability between two distinct processes is always assumed *bidirectional*: given two process p and q, with $p \neq q$, p can communicate with q if and only if q can also communicate with p.

Each process p distinguish all its neighbors using local labels. The set of local labels at p is denoted by $\mathcal{N}(p)$. For every neighbor q of p, we assume that q knows its local label in $\mathcal{N}(p)$. By abuse of notation, we denote the local label of q, at any of its neighbors, by q.

Hence, we represent the communication network *topology* of a distributed system by an undirected graph $G = (V, E)$ where V is the set of processes and E is a set of edges representing the communicating relation between processes: Unless explicitly mentioned, we always assume that the graph is *connected*.

3.3 Locally Shared Memory Model

3.3.1 Local Program

We assume the *locally sharing variables* introduced by Dijkstra [Dij74], where each process owns a finite set of *variables*. Communication is carried out by these variables as follows: Every process p can read its own variables and the variables of its neighbors, but it can only write its own variables.

The *local program* of a process p is defined by a finite set of *actions* (or guarded commands) which are written as follows:

$$\langle label \rangle \ :: \ \langle guard \rangle \ \longrightarrow \ \langle statement \rangle$$

The *label* of an action is its identifier. The *guard* of an action of a process p is a Boolean expression involving the variables of p and its neighbors. The *statement* of an action of p updates one or more variables of p. We assume the model of *composite atomicity* [DIM93], that is, guard evaluation and statement execution are assumed to take place in a single atomic (*i.e.*, uninterrupted) step.

An action can be executed if and only if its guard evaluates to TRUE, we then say that this action is *enabled*. By extension, a process is said to be *enabled* if and only if at least one of its actions is enabled.

Priorities. In order to simplify the presentation of a local program, we give a distinct *priority* to each action, by writing it as a number in parenthesis in front of its label. Then, an action is enabled if and only if its guard evaluates to TRUE and there is no other enabled action of higher priority.

Note that introducing priorities only eases local program readability. We show hereafter a simple method that rewrites any set of actions with priorities to an equivalent one without priorities.

Let $< \ell_1 :: g_1 \longrightarrow s_1, \ldots, \ell_x :: g_x \longrightarrow s_x >$ be a list of x actions ordered by priorities, *i.e.*, $\forall i, j \in [1..x], i < j \Leftrightarrow$ the action of label l_i has an higher priority than the action of label l_j. Note that the action of label l_1 is of highest priority.

We denote by $\{\ell'_1 :: g'_1 \longrightarrow s'_1, \ldots, \ell'_x :: g'_x \longrightarrow s'_x >$ the resulting set of x actions without priorities, where $\forall i \in [1..x]$, we have:

- $l'_i = l_i$,
- $g'_i = g_i \wedge \bigwedge_{j=1}^{i-1} \neg g_j$ (in particular, $g'_1 = g_1$), and
- $s'_i = s_i$.

In the guards of the latter set of actions, the conjunction of the negation of guards of higher priority prevents the corresponding action to be enabled when the guard of an action of higher priority evaluates to TRUE. Labels and statements remain unchanged.

3.3.2 Distributed Algorithm

A *distributed algorithm* is a collection of n local programs, each one operating on a single process. Note that some distributed algorithms may be designed for a restricted set of topologies $\mathcal{G}$. Let $\mathcal{A}$ be a distributed algorithm. For every process p, we denote by $\mathcal{A}(p)$ the local program of p in $\mathcal{A}$.

State and Configuration. Let p be a process. The *state* of process p in the local program $\mathcal{A}(p)$ is defined by the values of its variables in $\mathcal{A}(p)$.

A *configuration* is an instance of the states of all processes in $\mathcal{A}$. A configuration is *terminal* if and only if no process is enabled in this configuration. We denote the set of all possible configurations by $\mathcal{C}_\mathcal{A}$ (or simply $\mathcal{C}$ when unambiguous). Let $\gamma \in \mathcal{C}$, then $\gamma(p)$ denotes the local state of process p in configuration γ.

Step. Let γ be a configuration, we denote the set of enabled processes in γ by $\mathit{Enabled}(\gamma) \in \mathcal{P}(V)$.

If $\mathit{Enabled}(\gamma) = \emptyset$, γ is terminal.

Otherwise ($\mathit{Enabled}(\gamma) \neq \emptyset$) a non-empty subset of processes, $\mathit{Activated}(\gamma)$, of $\mathit{Enabled}(\gamma)$ is *activated*. Each process of $\mathit{Activated}(\gamma)$ *atomically* executes its enabled action of highest priority, leading then to a new configuration γ'. Such a transition from γ to γ' is called a *step* and denoted by $\gamma \mapsto \gamma'$. Note that $(\mathcal{C}, \mapsto)$ is a transition system over the configurations of $\mathcal{A}$.

The set $\mathit{Enabled}(\gamma)$ of enabled processes in configuration γ is computed by evaluating the guards of $\mathcal{A}$. The set $\mathit{Activated}(\gamma)$ of activated processes in configuration γ is selected by a daemon defined hereafter. Daemon materializes the asynchronism of the system: every process may run at different speeds.

3.3.3 Daemon

Let $\mathcal{A}$ be a distributed algorithm and $\mathcal{G}$ a set of topologies.

We assume that each step $\gamma \mapsto \gamma'$, from a configuration of $\mathcal{C}_\mathcal{A}$ to another, is driven by a *daemon*, that is an adversary which selects the set $Activated(\gamma)$ of processes that are activated, as a subset of $Enabled(\gamma)$.

Definition 1 (Daemon) *A* daemon *is a function $\mathfrak{d} : \mathcal{C}^+ \to \mathcal{P}(V)$ such that, given any finite sequence of configurations $(\gamma_0\gamma_1 \dots \gamma_i) \in \mathcal{C}^i$ through which the system has evolved, we have $\mathfrak{d}(\gamma_0\gamma_1 \dots \gamma_i) = Activated(\gamma_i) \subseteq Enabled(\gamma_i)$.*

We denote by $\mathfrak{D}_{all}$ the set of all daemons.

As the local program of every process is deterministic, we also represent algorithm $\mathcal{A}$ as a function $f_\mathcal{A} : \mathcal{C} \times \mathcal{P}(V) \to \mathcal{C}$. Note that, $\forall \gamma \in \mathcal{C}$, $f_\mathcal{A}(\gamma, \emptyset) = \gamma$.

Definition 2 (Execution) *Let $\mathfrak{d}$ be a daemon. An* execution *of $\mathcal{A}$ in any instance of $\mathcal{G}$ under $\mathfrak{d}$ is a* maximal *sequence of its configurations $e = (\gamma_0\gamma_1 \dots \gamma_i \dots) \in \mathcal{C}^\omega$ inductively defined as follows:*

1. *$\gamma_0 \in \mathcal{C}$;*
2. *$\forall i \geq 0$, $\gamma_{i+1} = f_\mathcal{A}(\gamma_i, \mathfrak{d}(\gamma_0\gamma_1 \dots \gamma_i))$, that is, γ_{i+1} is obtained from γ_i by an atomic step of all processes in $\mathfrak{d}(\gamma_0\gamma_1 \dots \gamma_i)$.*

Here, the term "maximal" means that the execution is either infinite, or ends at a terminal *configuration.*

Given an algorithm $\mathcal{A}$, a set of topologies $\mathcal{G}$, and a family of daemons $\mathfrak{D}$, we denote the set of all possible executions of $\mathcal{A}$ in any instance of $\mathcal{G}$ under any instance of $\mathfrak{D}$ by $\mathcal{E}_{\mathcal{A},\mathcal{G},\mathfrak{D}}$ (or simply $\mathcal{E}$ when unambiguous).

Definition 3 (Proper Daemon) *Let $\mathfrak{d}$ be a daemon. The daemon $\mathfrak{d}$ is* proper *if and only if, for every $i \geq 0$, when one or more processes are enabled in configuration γ_i, the daemon selects at least one of these enabled processes to execute an action, that is, for every execution $e = (\gamma_0\gamma_1 \dots \gamma_i \dots)$ of algorithm $\mathcal{A}$ in any instance of $\mathcal{G}$ under the daemon $\mathfrak{d}$, we have for every $i \geq 0$, $Enabled(\gamma_i) \neq \emptyset \Rightarrow \mathfrak{d}(\gamma_0\gamma_1 \dots \gamma_i) \neq \emptyset$.*

We denote by $\mathfrak{D}_P$ the set of proper daemons.

In the following, we first present some properties of *fairness* and *distribution* which are defined for proper daemons. Then we give a classification of daemons based on these properties.

3.3.3.1 Fairness

The *fairness* of a daemon matters how often it may prevent a process from being activated while enabled. Given any non-terminal configuration, by definition, proper daemons cannot prevent all enabled processes from being activated, that is, they cannot completely prevent algorithm's progress. This *progress property* is usually

required by deterministic algorithms. When there is no other restriction on its fairness, a proper daemon is said to be *unfair*, that is, it can forever prevent a process to execute an action, except if this process is the only enabled process. Hereafter, we present the main fairness assumptions on a proper daemon.

Definition 4 (Weakly Fair Daemon) *A proper daemon is* weakly fair *if and only if it eventually allows every* continuously *enabled process to execute an action.*

Given an algorithm $\mathcal{A}$, a set of topologies $\mathcal{G}$, and a daemon $\mathfrak{d} \in \mathfrak{D}_{all}$, $\mathfrak{d}$ is weakly fair *if and only if $\mathfrak{d}$ is proper and for every execution $e = (\gamma_0\gamma_1 \dots \gamma_i \dots)$ of algorithm $\mathcal{A}$ in any instance of $\mathcal{G}$ under the daemon $\mathfrak{d}$, $\forall p \in V$, $\forall i \geq 0$, $(p \in Enabled(\gamma_i) \Rightarrow \exists j \geq i : \ p \notin Enabled(\gamma_j) \vee p \in \mathfrak{d}(\gamma_0\gamma_1 \dots \gamma_j))$.*

We denote by $\mathfrak{D}_{WF}$ the family of weakly fair daemons.

Definition 5 (Strongly Fair Daemon) *A proper daemon is* strongly fair *if it allows every* infinitely often *enabled process to execute infinitely often many actions.*

Given an algorithm $\mathcal{A}$, a set of topologies $\mathcal{G}$, and a daemon $\mathfrak{d} \in \mathfrak{D}_{all}$, $\mathfrak{d}$ is strongly fair *if and only if $\mathfrak{d}$ is proper and for every execution $e = (\gamma_0\gamma_1 \dots \gamma_i \dots)$ of algorithm $\mathcal{A}$ in any instance of $\mathcal{G}$ under the daemon $\mathfrak{d}$, for every process p, we have: $(\forall i \geq 0, \exists j \geq i : p \in Enabled(\gamma_j)) \Rightarrow (\forall i' \geq 0, \exists j' \geq i' : p \in \mathfrak{d}(\gamma_0\gamma_1 \dots \gamma_{j'}))$.*

We denote by $\mathfrak{D}_{SF}$ the family of strongly fair daemons.

3.3.3.2 Distribution

The *distribution* of a proper daemon matters how many enabled processes it may activate or prevent from being activated simultaneously. When there is no restriction on its distribution, a proper daemon is *distributed*, that is, it can activate any non-empty subset of enabled processes. Hereafter, we present the main distribution assumptions on a proper daemon.

Definition 6 (Central Daemon) *A daemon is* central *if and only if exactly one enabled process is activated at each step.*

Given an algorithm $\mathcal{A}$, a set of topologies $\mathcal{G}$, and a daemon $\mathfrak{d} \in \mathfrak{D}_{all}$, $\mathfrak{d}$ is central *if and only if $\mathfrak{d}$ is proper and for every execution $e = (\gamma_0\gamma_1 \dots \gamma_i \dots)$ of algorithm $\mathcal{A}$ in any instance of $\mathcal{G}$ under the daemon $\mathfrak{d}$, for every $i \geq 0$, $|\mathfrak{d}(\gamma_0\gamma_1 \dots \gamma_i)| \leq 1$.*

We denote by $\mathfrak{D}_C$ the family of central daemons.

Definition 7 (Synchronous Daemon) *A daemon is* synchronous *if and only if every enabled process is activated at each step.*

Given an algorithm $\mathcal{A}$, a set of topologies $\mathcal{G}$, and a daemon $\mathfrak{d} \in \mathfrak{D}_{all}$, $\mathfrak{d}$ is synchronous *if and only if for every execution $e = (\gamma_0\gamma_1 \dots \gamma_i \dots)$ of algorithm $\mathcal{A}$ in any instance of $\mathcal{G}$ under the daemon $\mathfrak{d}$, for every $i \geq 0$, $\mathfrak{d}(\gamma_0\gamma_1 \dots \gamma_i) = Enabled(\gamma_i)$.*

We denote by $\mathfrak{D}_S$ the family of synchronous daemons.

3.3.3.3 Classification

Here, we give a classification of the daemons we are interested in, that is the main proper daemons.

Note that, unless a fairness property is specified, daemons are considered to be unfair, which is the weakest fairness assumption for proper daemons. Likewise, unless a distribution property is specified, daemons are considered to be distributed, which is the weakest distribution assumption for proper daemons.

We recall the families of daemons previously defined in this section:

- $\mathfrak{D}_{all}$, the set of all daemons.
- $\mathfrak{D}_P = \{\mathfrak{d} \in \mathfrak{D}_{all}, \mathfrak{d} \text{ is proper}\}$
  $= \{\mathfrak{d} \in \mathfrak{D}_P, \mathfrak{d} \text{ is distributed and unfair}\} = \mathfrak{D}_{D,U}$
- $\mathfrak{D}_{WF} = \{\mathfrak{d} \in \mathfrak{D}_P, \mathfrak{d} \text{ is weakly fair}\}$
  $= \{\mathfrak{d} \in \mathfrak{D}_P, \mathfrak{d} \text{ is distributed and weakly fair}\} = \mathfrak{D}_{D,WF}$
- $\mathfrak{D}_{SF} = \{\mathfrak{d} \in \mathfrak{D}_P, \mathfrak{d} \text{ is strongly fair}\}$
  $= \{\mathfrak{d} \in \mathfrak{D}_P, \mathfrak{d} \text{ is distributed and strongly fair}\} = \mathfrak{D}_{D,SF}$
- $\mathfrak{D}_C = \{\mathfrak{d} \in \mathfrak{D}_P, \mathfrak{d} \text{ is central}\}$
  $= \{\mathfrak{d} \in \mathfrak{D}_P, \mathfrak{d} \text{ is central and unfair}\} = \mathfrak{D}_{C,U}$
- $\mathfrak{D}_S = \{\mathfrak{d} \in \mathfrak{D}_P, \mathfrak{d} \text{ is synchronous}\}$

From the aforementioned families of daemons, we define the following additional families of daemons by mixing fairness and distribution assumptions:

- $\mathfrak{D}_{C,WF} = \{\mathfrak{d} \in \mathfrak{D}_P, \mathfrak{d} \text{ is central and weakly fair}\}$
- $\mathfrak{D}_{C,SF} = \{\mathfrak{d} \in \mathfrak{D}_P, \mathfrak{d} \text{ is central and strongly fair}\}$

Set relations between these families of proper daemons are shown in Figure 3.1 where every family is represented. We can remark the following set relations:

- $\mathfrak{D}_S \subseteq \mathfrak{D}_{SF}$
- $\mathfrak{D}_{SF} \subseteq \mathfrak{D}_{WF} \subseteq \mathfrak{D}_P$
- $\mathfrak{D}_C \subseteq \mathfrak{D}_P$
- $\mathfrak{D}_{C,WF} = \mathfrak{D}_C \cap \mathfrak{D}_{WF}$
- $\mathfrak{D}_{C,SF} = \mathfrak{D}_C \cap \mathfrak{D}_{SF}$

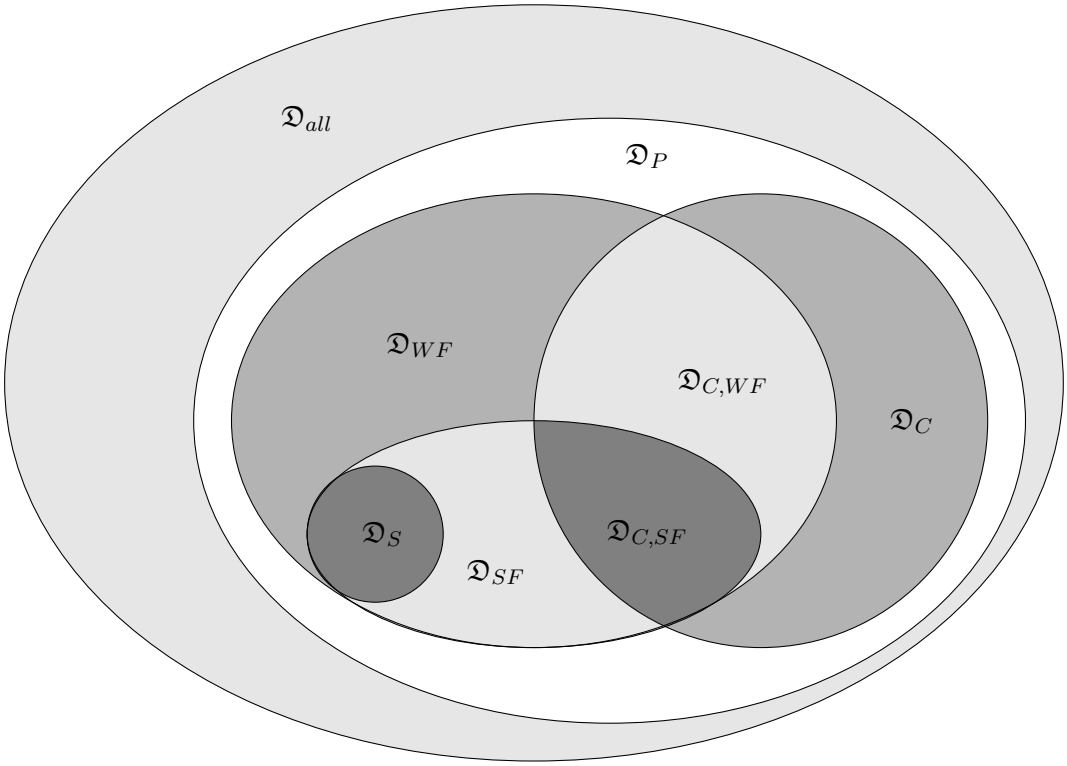


Figure 3.1 – Illustration of daemons families, emphasizing set relations between them.

From these set relations, note that a central daemon is a stronger assumption than a distributed daemon of same or weaker fairness, because it cannot activate more than one enabled process at each step.

A synchronous daemon is a stronger assumption than a distributed daemon, as it cannot prevent any enabled process from being activated.

The distribution assumption of a synchronous daemon is also a kind of fairness assumption and even the strongest one, since a synchronous daemon has to activate every process each time they are enabled.

Finally, synchronous and central daemons cannot be compared, they can generate the same execution for specific topologies and algorithms such that only one process is enabled at each step, but we usually do not make any assumption on the number of enabled processes.

3.4 Self-Stabilization and Related Properties

In this subsection, we are interested in properties over distributed algorithm which have been defined to the purpose of fault-tolerance, particularly the tolerance of transient faults. Note that transient faults may corrupt the state of processes, but they are assumed not to alter their local program.

3.4.1 Self-Stabilization

Let $\mathcal{A}$ be a distributed algorithm, $\mathcal{G}$ a set of topologies, and $\mathfrak{D}$ a family of daemons. We recall that $\mathcal{E}$ denotes the set of all possible executions of $\mathcal{A}$ in any instance of $\mathcal{G}$ under any daemon of $\mathfrak{D}$. We denote by $\mathcal{E}_\gamma$ the set of all possible executions – of $\mathcal{A}$ in any instance of $\mathcal{G}$ under any daemon of $\mathfrak{D}$ – which start from configuration γ.

A *specification* is a predicate *Spec* over $\mathcal{E}$, or equivalently, a subset $\mathcal{E}'$ of $\mathcal{E}$ such that $\forall e \in \mathcal{E}, e \in \mathcal{E}' \Leftrightarrow Spec(e)$.

Let *Spec* be a specification. Algorithm $\mathcal{A}$ *is self-stabilizing with respect to Spec* in any instance of $\mathcal{G}$ under any daemon of $\mathfrak{D}$, if and only if there exists a non-empty subset $\mathcal{L}$ of $\mathcal{C}$ such that:

Closure: For every configuration γ in $\mathcal{L}$, for every execution e_γ in $\mathcal{E}_\gamma$, $Spec(e_\gamma) =$ TRUE.

Convergence: For every execution e in $\mathcal{E}_{\mathcal{A},\mathcal{G},\mathfrak{D}}$, there is a configuration γ in e such that γ is in $\mathcal{L}$.

The configurations of $\mathcal{L}$ are said to be *legitimate*, and other configurations are called *illegitimate*.

In other words, algorithm $\mathcal{A}$ is self-stabilizing with respect to specification *Spec* – in any instance of $\mathcal{G}$ under any daemon of $\mathfrak{D}$ – if and only if, starting from any configuration, it reaches, in finite time, a legitimate configuration from which every execution of $\mathcal{A}$ satisfies *Spec*.

A specification *Spec* is *static*, if and only if there is a predicate P over $\mathcal{C}$, such that, for every execution e in $\mathcal{E}_{\mathcal{A},\mathcal{G},\mathfrak{D}}$, $Spec(e)$ if and only if $\forall \gamma \in e, P(\gamma)$.

We can reformulate the above definition of self-stabilization for any static specification. Let $\mathcal{A}$ be a distributed algorithm, $\mathcal{G}$ a set of topologies, $\mathfrak{D}$ a family of daemons, and P be a predicate over the set $\mathcal{C}$ of all possible configurations of $\mathcal{A}$. $\mathcal{A}$ *is self-stabilizing with respect to* P – in any instance of $\mathcal{G}$ under any daemon of $\mathfrak{D}$ – if and only if there is a non-empty subset $\mathcal{L}$ of $\mathcal{C}$ such that:

Correction: $\forall \gamma \in \mathcal{L}$, $P(\gamma)$.

Closure: $\forall \gamma_i \mapsto \gamma_{i+1}$, $\gamma_i \in \mathcal{L} \Rightarrow \gamma_{i+1} \in \mathcal{L}$.

Convergence: $\forall e \in \mathcal{E}$, $\exists \gamma \in e$, $\gamma \in \mathcal{L}$.

Closure and convergence properties are shown in Figure 3.2. Vertices of the graph represent some configurations of the system and edges represent all possible steps between these configurations.

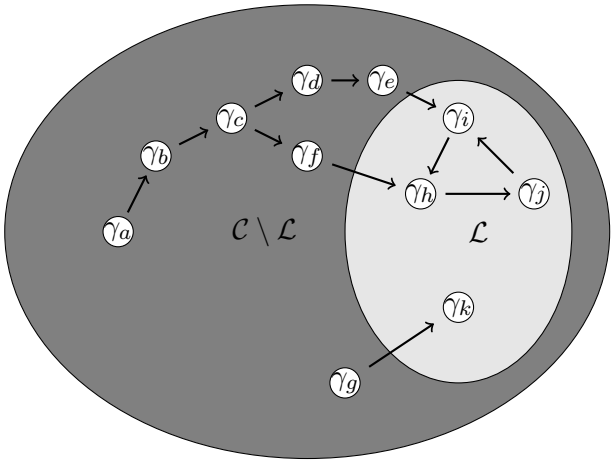


Figure 3.2 – Self-stabilization in the space of system configurations.

3.4.2 Silence

Let $\mathcal{A}$ be a distributed algorithm, $\mathcal{G}$ a set of topologies, and $\mathfrak{D}$ a family of daemons. We say that algorithm $\mathcal{A}$ is *silent* [DGS96], in any instance of $\mathcal{G}$, under any daemon of $\mathfrak{D}$, if and only if every execution in $\mathcal{E}_{\mathcal{A},\mathcal{G},\mathfrak{D}}$ is finite. In other words, starting from an arbitrary configuration, the network will eventually reach a terminal configuration.

Let P be a predicate over the set of configurations of $\mathcal{A}$. To show that algorithm $\mathcal{A}$ is silent and self-stabilizing with respect to predicate P in any instance of $\mathcal{G}$ under a daemon of $\mathfrak{D}$, it is sufficient to show that:

1. Every execution in $\mathcal{E}_{\mathcal{A},\mathcal{G},\mathfrak{D}}$ is finite; and

2. Every terminal configuration of $\mathcal{A}$ satisfies P.

3.4.3 Space Complexity

The space (memory) complexity of an algorithm $\mathcal{A}$ is computed from the number of bits required to represent every possible states of $\mathcal{A}(p)$ for every process p. The bits requirement for representing a state at any process p is the sum of bits requirements for every variable of $\mathcal{A}(p)$.

We assume, as usual, that a variable with x distinct possible values can be represented using $\lceil \log x \rceil$ bits. Moreover, we assume that the unique identifier (UID) of every process – which can take n distinct values – is represented by $O(\log n)$ bits.

3.4.4 Time Complexity

Let $\mathcal{A}$ be a self-stabilizing algorithm with respect to a specification *Spec* in any instance of $\mathcal{G}$ under any daemon of $\mathfrak{D}$. The *stabilization time* of $\mathcal{A}$ is the maximum time it takes to reach a legitimate configuration with respect to *Spec* starting from any configuration of $\mathcal{C}_{\mathcal{A}}$. It is expressed either as the number of steps or as the number of *rounds* (defined hereafter) for $\mathcal{A}$ to reach a legitimate configuration with respect to *Spec*.

We say that a process p is *neutralized* in the step $\gamma_i \mapsto \gamma_{i+1}$ if p is enabled in γ_i and not enabled in γ_{i+1}, but does not execute any action between these two configurations. The neutralization of a process represents the following situation: at least one neighbor of p changes its state between γ_i and γ_{i+1}, and this change effectively makes the guard of all actions of p FALSE.

To evaluate time complexity, we use the notion of *round*. The first round of an execution e, noted e', is the minimal prefix of e in which every process that is enabled in the initial configuration either executes an action or becomes neutralized. Let e'' be the suffix of e starting from the last configuration of e'. The second round of e is the first round of e'', and so forth.

3.4.5 Safe Convergence

Self-stabilizing algorithms guarantee to self-stabilizes in finite time with respect to a specification, but they do not give any guarantee related to this specification before convergence is achieved. Safe convergence is an attempt to address this drawback by enforcing algorithms to quickly self-stabilize with respect to a weaker specification, before seeking for long-term self-stabilization with respect to the original specification.

Let $\mathcal{A}$ be a distributed algorithm, $\mathcal{G}$ a set of topologies, $\mathfrak{D}$ a family of daemons, and P_1 and P_2 two predicates over $\mathcal{C}_{\mathcal{A}}$ such that $\forall \gamma \in \mathcal{C}_{\mathcal{A}}$, $P_2(\gamma) \Rightarrow P_1(\gamma)$. $\mathcal{A}$ is *safely converging self-stabilizing with respect to* (P_1, P_2) if and only if the following three properties hold:

(1) $\mathcal{A}$ is *self-stabilizing w.r.t.* P_1;

(2) $\mathcal{A}$ is *self-stabilizing w.r.t.* P_2; and

(3) Every execution of $\mathcal{A}$ – in any instance of $\mathcal{G}$ under any daemon of $\mathfrak{D}$ – starting from a configuration of $\mathcal{L}_{P_1}$ eventually reaches a configuration of $\mathcal{L}_{P_2}$, where $\mathcal{L}_{P_1}$ and $\mathcal{L}_{P_2}$ are respectively the sets of legitimate configurations for P_1 and P_2.

The configurations of $\mathcal{L}_{P_1}$ are said to be *feasible legitimate*. The configurations of $\mathcal{L}_{P_2}$ are said to be *optimal legitimate*.

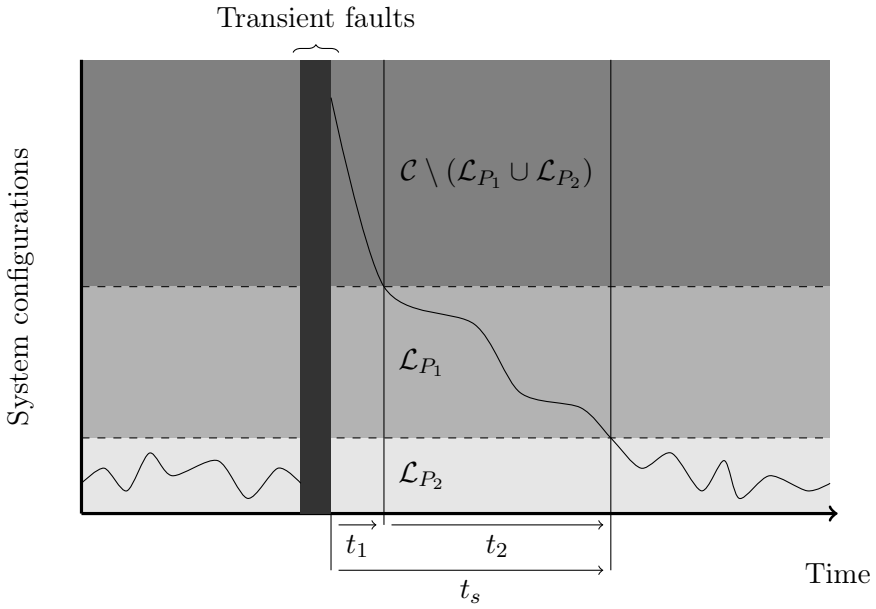


Figure 3.3 – Safely converging self-stabilization of a system, where t_1 is the first convergence time, t_2 is the second convergence time, and t_1 is the stabilization time.

Assume that $\mathcal{A}$ is *safely converging self-stabilizing w.r.t.* (P_1, P_2) in any instance of $\mathcal{G}$ under any daemon of $\mathfrak{D}$. The *first convergence time* is the maximum time to reach a feasible legitimate configuration, starting from any configuration. The *second convergence time* is the maximum time to reach an optimal legitimate configuration, starting from any feasible legitimate configuration. The *stabilization time* is the sum of the first and second convergence times.

> In the following, when we write that an algorithm self-stabilizes w.r.t. some specification "under a particular daemon", it means "under any daemon of that family of particular daemons". Besides, when we omit to precise the set of graphs for which an algorithm is self-stabilizing, it means "in any connected graph".

3.5 Hierarchical Collateral Composition

Composition techniques are often used to simplify the design and the proofs of self-stabilizing algorithms [Tel01]. Lots of composition techniques have been proposed so far, among them, the *collateral composition* introduced by Herman [Her92] and the *fair composition* introduced by Dolev [Dol00]. These two approaches are really closed. In the collateral composition, the composition of two algorithms just consists of running the two algorithms concurrently, the second algorithm using the output of the first one in its computations. Now, when two actions are enabled at the same process but in two different composed algorithms, the process nondeterministically executes one or the other, if activated by the daemon. This nondeterminism is solved

in the fair composition as follows: each process runs the composed algorithms in alternation.

Here, we use a slightly modified version of the *collateral composition* [Her92], in which we solve the nondeterminism of the collateral composition as follows: When we compose two distributed algorithms $\mathcal{A}$ and $\mathcal{B}$, we modify the code of $\mathcal{B}(p)$ (for every process p) so that p executes an action of $\mathcal{B}(p)$ only when it has no enabled action in $\mathcal{A}(p)$.

Definition 8 (Hierarchical Collateral Composition) *Let $\mathcal{A}$ and $\mathcal{B}$ be two (distributed) algorithms such that no variable written by $\mathcal{B}$ appears in $\mathcal{A}$. In the* hierarchical collateral composition *of $\mathcal{A}$ and $\mathcal{B}$, noted $\mathcal{B} \circ \mathcal{A}$, the (local) program of every process p, $\mathcal{B}(p) \circ \mathcal{A}(p)$, is defined as follows:*

- *$\mathcal{B}(p) \circ \mathcal{A}(p)$ contains all variables of $\mathcal{A}(p)$ and $\mathcal{B}(p)$.*
- *$\mathcal{B}(p) \circ \mathcal{A}(p)$ contains all actions of $\mathcal{A}(p)$.*
- *For every action $(l :: g \longrightarrow s)$ of $\mathcal{B}(p)$, $\mathcal{B}(p) \circ \mathcal{A}(p)$ contains the action $(l' :: (\neg dis_{\mathcal{A}(p)} \wedge g) \longrightarrow s$ where $dis_{\mathcal{A}(p)}$ is the disjunction of all guards of actions in $\mathcal{A}(p)$.*

Below, we give two properties of the *hierarchical collateral composition*: Theorem 1 and Corollary 1. Corollary 1 states a sufficient condition to show the correctness of the composite algorithm. To prove these properties, we need to first define the notions of *minimal relevant subsequence* and *projection*.

Definition 9 (MRS) *Let s be a sequence of configurations. The* minimal relevant subsequence *of s, noted $\mathcal{MRS}(s)$, is the maximal subsequence of s where no two consecutive configurations are identical.*

Definition 10 (Projection) *Let γ be a configuration and $\mathcal{A}$ be an algorithm. The* projection *$\gamma_{|\mathcal{A}}$ is the configuration obtained by removing from γ the values of all variables that do not exist in $\mathcal{A}$. Let $e = \gamma_0\gamma_1 \ldots \gamma_i$ be a sequence of configurations, the* projection *$e_{|\mathcal{A}}$ is the sequence $\gamma_{0|\mathcal{A}}\gamma_{1|\mathcal{A}} \cdots \gamma_{i|\mathcal{A}}$.*

Roughly speaking, the following theorem shows that if $\mathcal{A}$ is a silent self-stabilizing algorithm in the composite algorithm $\mathcal{B} \circ \mathcal{A}$, and the daemon is weakly fair, then $\mathcal{B}$ cannot prevent $\mathcal{A}$ to reach a legitimate terminal configuration.

Theorem 1 *Let $\mathcal{A}$ be a silent algorithm that stabilizes with respect to $Spec_{\mathcal{A}}$ under a weakly fair daemon. Let $\mathcal{B}$ be an algorithm such that no variable written by $\mathcal{B}$ appears in $\mathcal{A}$. $\mathcal{B} \circ \mathcal{A}$ satisfies the two following claims:*

1. *It stabilizes with respect to $Spec_{\mathcal{A}}$ under a weakly fair daemon.*
2. *It eventually reaches a configuration where no action of $\mathcal{A}$ is enabled ever.*

Proof. Let an execution e of $\mathcal{B} \circ \mathcal{A}$ under the weakly fair daemon. Let $e' = \mathcal{MRS}(e_{|\mathcal{A}})$. No variable in the configurations of e' are written by $\mathcal{B}$ and all configurations of e' are possible configurations of $\mathcal{A}$.

Consider any processor p continuously enabled w.r.t. algorithm $\mathcal{A}$ in a configuration γ of e'. Then, by construction p is continuously enabled to execute an action of $\mathcal{A}$ from the first configuration of e that generates γ, thus it eventually executes an action of $\mathcal{A}$ in e and consequently in e'. So, e' is a possible execution of $\mathcal{A}$ under the weakly fair daemon. Consequently, e' stabilizes with respect to $Spec_{\mathcal{A}}$ and is finite. Hence, e also stabilizes with respect to $Spec_{\mathcal{A}}$ and eventually reaches a configuration where no action of $\mathcal{A}$ is enabled ever. □

From the previous theorem, we immediately deduce the following corollary:

Corollary 1 *$\mathcal{B} \circ \mathcal{A}$ stabilizes with respect to Spec under a weakly fair daemon if the following conditions hold:*

1. *$\mathcal{A}$ is a silent (self-stabilizing) algorithm under a weakly fair daemon.*
2. *$\mathcal{B}$ stabilizes under a weakly fair daemon to Spec from any configuration where no action of $\mathcal{A}$ is enabled ever.*[1]

Proof. By Theorem 1.(2) and (1), any execution of $\mathcal{B} \circ \mathcal{A}$ assuming a weakly fair daemon reaches a configuration γ from which no action of $\mathcal{A}$ is enabled ever. Then, from γ, $\mathcal{B}$ stabilizes with respect to *Spec* by (2). □

3.6 Fairness Transformer

We give an automatic method for transforming any self-stabilizing algorithm which works under a weakly fair daemon into a self-stabilizing algorithm which works under an unfair daemon (for the same specification). Our method preserves the silence property of the input algorithm.

There already exist several methods to transform a weakly fair algorithm into an unfair one. In [BGJ01], the authors define the *cross-over* composition. Using this composition, a weakly fair algorithm can be transformed by composing it with an algorithm that is fair under an unfair daemon. However, this technique does not preserve the silence of the input algorithm. Moreover, no step complexity analysis is given for the output unfair algorithm. In [KK06], authors give a transformer that preserves the silence of the input algorithm. Furthermore, the step complexity the transformed algorithm is $O(n^4 \times R)$, where R is the stabilization time of the input algorithm in rounds. Finally, note that the round complexity of the transformed version is much higher than that of the input algorithm (of the same order as the step complexity).

In contrast with the previous solutions, our transformer does not degrade the round complexity of the algorithm. Moreover, the step complexity of the transformed

[1] Recall that in such a configuration, the specification of $\mathcal{A}$ is satisfied.

algorithm is $O(\mathcal{D}n(R+n^2))$, where R is the stabilization time of the input algorithm in rounds. For a case study, please refer to Algorithm $\mathcal{SMDS}(k)$ in Chapter 7.

Let $\mathcal{A}$ be an algorithm that stabilizes w.r.t. $Spec_{\mathcal{A}}$, assuming a weakly fair daemon.[2] Let p be a process. We recall that $\mathcal{A}(p)$ denotes the local program of p in $\mathcal{A}$. Assume that $\mathcal{A}(p)$ has x actions. Actions of $\mathcal{A}(p)$ are indexed by $[0..x-1]$, and are of the following form:

$$A_i \ :: \ G_i \ \longrightarrow \ S_i.$$

We denote by $\mathcal{A}^t$ the transformed version of $\mathcal{A}$. $\mathcal{A}^t$ is obtained by composing $\mathcal{A}$ with a self-stabilizing phase clock algorithm. This latter algorithm, called $\mathcal{U}$, is treated as a black box ($\mathcal{U}(p)$ denotes the local program of p in $\mathcal{U}$), with the following properties:

1. Every process p has an incrementing variable $p.clock \in \mathbb{Z}_\alpha$, the cyclic group of order α, where $\alpha \geq 3$
2. The phase clock is self-stabilizing, assuming an unfair daemon, *i.e.*, after it has stabilized, there exists an integer function f on processes such that:
   - $f(p) \bmod \alpha = p.clock$
   - For all processes p and q, $|f(p) - f(q)| \leq \|p, q\|$.
   - For every process p, $f(p)$ increases by 1 infinitely often using statement $Incr(p)$.
3. Every process p has in its local program $\mathcal{U}(p)$ an action $I :: Can\_Incr(p) \rightarrow Incr(p)$ such that, once $\mathcal{U}$ is stabilized, I is the only action that p is enabled to execute. Moreover, $\mathcal{U}$ does not require execution of Action I during the stabilization phase.

An algorithm that matches all these requirements can be found in [BPV04].

The local program of each process p in $\mathcal{A}^t$ is obtained as follows:

- $\mathcal{A}^t(p)$ contains all variables of $\mathcal{A}(p)$ and $\mathcal{U}(p)$.
- $\mathcal{A}^t(p)$ contains all actions of $\mathcal{U}(p)$, except I, which is replaced by the following actions:
  - $A'_i :: Can\_Incr(p) \wedge G_i \rightarrow (Incr(p); S_i)$ for every $i \in [0..x-1]$,
  - $L :: Can\_Incr(p) \wedge Stable_p \wedge Late_p \rightarrow Incr(p)$ where $Stable_p \equiv (\forall i \in [0..x-1] : \neg G_i)$ and $Late_p \equiv \neg(\forall q \in \mathcal{N}(p) : q.clock = p.clock)$

Roughly speaking, our transformer enforces fairness among processes that are enabled in $\mathcal{A}$ because they can only move once at each clock tick. Once $\mathcal{A}$ has stabilized, if $\mathcal{A}$ is silent, then every process p eventually satisfies $Stable_p$ and, once all clocks have the same value, no further action is enabled, hence the silence is preserved.

[2] In particular, if $\mathcal{A}$ is silent, any configuration of $\mathcal{A}$ satisfying $Spec_{\mathcal{A}}$ is terminal.

Theorem 2 *$\mathcal{A}^t$ stabilizes with respect to $Spec_{\mathcal{A}}$ under an unfair daemon.*

Proof. By construction, any execution of $\mathcal{A}^t$ converges to a configuration γ' that is legitimate w.r.t. algorithm $\mathcal{U}$. Consider any configuration γ'' reachable from γ'. Assume that $\bigvee_{i\in[0..x-1]} G_i$ continuously holds at process p from γ'' but p never again executes any A'_i. $Stable_p$ is FALSE forever from γ'' and, consequently, $p.clock$ is never again incremented. As $\mathcal{U}$ works under an unfair daemon, eventually every process $q \neq p$ is disabled. In this case, $f(p)$ is minimum in the system. In particular, $Can\_Incr(p)$ holds. Thus, p is enabled to execute some A'_i. Hence, p is the only enabled process and it executes one of its enabled actions A'_i in the next step. Thus, if $\bigvee_{i\in[0..x-1]} G_i$ continuously holds at p from γ'', then p eventually executes one of its enabled actions A'_i in $\mathcal{A}^t$. As $\mathcal{A}$ stabilizes under a weakly fair daemon, $\mathcal{A}^t$ stabilizes w.r.t. the same specification under an unfair daemon. □

Theorem 3 *If $\mathcal{A}$ is silent, then $\mathcal{A}^t$ is silent.*

Proof. First, by Theorem 2 (and its proof), $\mathcal{A}^t$ converges to a configuration γ from which both the specification of algorithm $\mathcal{U}$ and the predicate $Stable_p$ for every process p hold forever. So, from γ, only Action L can be executed by processes. Let $M = \max_{p\in V} f(p)$, and $m = \min_{p\in V} f(p)$. While $M \neq m$, only processes q such that $f(q) \neq M$ could be enabled to execute Action L. Moreover, when executing Action L, any q increases $f(q)$ by 1. Hence, eventually, $M = m$ and no action is ever again enabled in the system. □

Below, we present the complexity of the transformed algorithm. These results assume that $\mathcal{U}$ is the algorithm of Boulinier *et al.* in [BPV04] The authors show that $2n-1$ states per process (actually the range of the phase clock) are sufficient to make $\mathcal{U}$ work in any topology (the worst case being the cycle topology). Moreover, using $2n-1$ states, the stabilization time of $\mathcal{U}$ is in $O(n)$ rounds [Bou07] and $O(\mathcal{D}n^3)$ steps [DP12], respectively. Hence, we have the following theorem:

Theorem 4 *The space complexity of $\mathcal{A}^t$ is $O(\log n) + MEM$ bits per process, where MEM is the memory requirement for $\mathcal{A}$.*

Below, we prove an additional result about $\mathcal{U}$:

Lemma 1 *Once $\mathcal{U}$ is stabilized, every process advances its local clock of $\mathcal{D}$ ticks at most every $2\mathcal{D}$ rounds.*

Proof. Let $f^{\min}_\gamma = \min_{p\in V} f(p)$ in some configuration γ after $\mathcal{U}$ stabilized. Let q be a process and f^q_γ be the value of $f(q)$ in γ. $f^{\min}_\gamma \leq f^q_\gamma \leq f^{\min}_\gamma + \mathcal{D}$. $2\mathcal{D}$ rounds after γ, $f(q) \geq f^{\min}_\gamma + 2\mathcal{D}$. Thus, $f(q) - f^q_\gamma \geq f^{\min}_\gamma + 2\mathcal{D} - (f^{\min}_\gamma + \mathcal{D})$, *i.e.*, $f(q) - f^q_\gamma \geq \mathcal{D}$. That is, q increments its phase clock at least $\mathcal{D}$ times during that period. □

Theorem 5 *$\mathcal{A}^t$ stabilizes with respect to $Spec_{\mathcal{A}}$ in $O(n + \lceil \frac{R}{\mathcal{D}} \rceil \times 2\mathcal{D})$ rounds, where R is the stabilization time of $\mathcal{A}$ in rounds, and if $\mathcal{A}$ is silent, then $\mathcal{A}^t$ reaches a terminal configuration in a round complexity of the same order of magnitude.*

Proof. First, $\mathcal{A}^t$ stabilizes w.r.t. the specification of $\mathcal{U}$ in $O(n)$ rounds. Then, $\mathcal{A}^t$ needs to emulate at most R rounds of $\mathcal{A}$ to stabilize w.r.t. $Spec_{\mathcal{A}}$. By Lemma 1, this requires at most $\lceil \frac{R}{\mathcal{D}} \rceil \times 2\mathcal{D}$ rounds.

Assume that $\mathcal{A}$ is silent. Then, consider the first configuration γ of $\mathcal{A}^t$ that is legitimate w.r.t. $Spec_{\mathcal{A}}$ and the specification of $\mathcal{U}$. Let $M = \max_{p \in V} f(p)$, and $m = \min_{p \in V} f(p)$ in γ. Then, $M - m \leq \mathcal{D}$. Hence, by Lemma 1, after at most $2\mathcal{D}$ additional rounds, $\mathcal{A}^t$ reaches a terminal configuration, and we are done. □

The next lemma gives a bound on the number of steps required to emulate a round of $\mathcal{A}$, once $\mathcal{U}$ has stabilized.

Lemma 2 *Once $\mathcal{U}$ has stabilized, every continuously enabled process in $\mathcal{A}^t$ executes an action after at most $2\mathcal{D}(n-1)$ steps.*

Proof. Consider a configuration γ after $\mathcal{U}$ has stabilized, and a process p that is continuously enabled from γ.

Then, $f(p) - \|p,q\| \leq f(q) \leq f(p) + \|p,q\|$ for every process $q \neq p$. So, every process $q \neq p$ can increment $q.clock$ at most $2\|p,q\|$ times before $p.clock$ is incremented. So, at most $\sum_{q \in V \setminus \{p\}} 2\|p,q\|$ steps can occur before p executes an action. As $\sum_{q \in V \setminus \{p\}} 2\|p,q\| \leq (n-1) \times 2\mathcal{D}$, the lemma holds. □

Theorem 6 *$\mathcal{A}^t$ stabilizes with respect to $Spec_{\mathcal{A}}$ in $O(\mathcal{D}n(R+n^2))$ steps, where R is the stabilization time of $\mathcal{A}$ in rounds; and if $\mathcal{A}$ is silent, then $\mathcal{A}^t$ reaches a terminal configuration, and its step complexity has the same order of magnitude.*

Proof. First, $\mathcal{A}^t$ stabilizes the specification of algorithm $\mathcal{U}$ in $O(\mathcal{D}n^3)$ steps. Then, by Lemma 2, we have that R rounds of $\mathcal{A}$ are emulated by $\mathcal{A}^t$ in $O(\mathcal{D}nR)$ steps.

Assume that $\mathcal{A}$ is silent. Then, consider the first configuration γ of $\mathcal{A}^t$ that is legitimate w.r.t. $Spec_{\mathcal{A}}$ and the specification of $\mathcal{U}$. Let $M = \max_{p \in V} f(p)$, and $m = \min_{p \in V} f(p)$ in γ. Then, $M - m \leq \mathcal{D}$. Hence, after $O(\mathcal{D}n)$ additional steps, $\mathcal{A}^t$ reaches a terminal configuration, and we are done. □

Part II

Support Structures

Chapter 4

Maximum Independent Set Tree

Contents

In this chapter, we first recall the definition of *MIS tree* (for Maximal Independent Set tree), introduced by Fernandess and Malkhi [FM02]. Next, we give a silent self-stabilizing algorithm that computes an *MIS tree* (for Maximal Independent Set Tree) in any arbitrary identified network within $O(n)$ rounds under a weakly fair daemon. There could be many different MIS trees for a given network and a given root r; the one we construct has the same specification as that constructed in [FM02], *i.e*, it is the *lexically first MIS tree*. We then prove the correctness of our algorithm, analyze its time and space complexity, and give an upper bound on the height of the constructed MIS tree. Finally, we show that the problem solved by our algorithm is $\mathcal{P}$-complete.

We make use of this data structure as a support for constructing a k-clustering of the network in Chapter 8.

4.1 Definition of MIS Tree

Let $G = (V, E)$ be a connected graph. An *MIS tree* (for Maximal Independent Set tree) of G is any spanning tree T of G rooted at some vertex r such that the set of vertices at even levels of T is a maximal independent set of G.

Property 1 *Let T be an MIS tree of a graph. Let I be the maximal independent set formed by the vertices at even levels of T. If σ is a path of T of length ℓ (*i.e.*, $\ell + 1$ vertices), then σ contains at least $\lceil \frac{\ell}{2} \rceil$ members of I.*

Assume that an ordering $p_1, p_2, \ldots, p_n$ of V is given. Any rooted tree T of G can be encoded as an n-tuple of numbers in the range $1..n$, as follows. The i^{th} entry of the encoding of T is j if p_j is the parent of p_i in T. The *lexically first MIS tree* (LFMIST) of G with root r is then defined to be that MIS tree of G whose encoding is first in the lexical order of the encodings of all MIS trees of G with root r. For example, in Figure 4.1, the members of the maximal independent set are shown in black and the encoding of the tree is $(1, 1, 2, 1, 3, 5, 8, 4, 6)$.

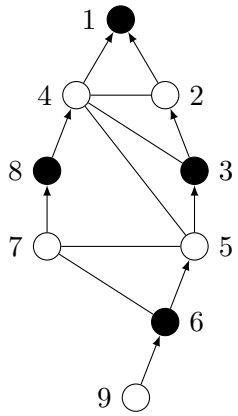


Figure 4.1 – Example of LFMIST (for lexically first maximal independent set tree).

4.2 Algorithm to construct an MIS Tree

We now give a silent self-stabilizing algorithm to construct an MIS tree (actually a LFMIST) in $O(n)$ rounds under a weakly fair. It is defined as the hierarchical collateral composition $\mathcal{MIST} \circ \mathcal{BFST}$, where $\mathcal{BFST}$ is a silent self-stabilizing algorithm that constructs a breadth-first search tree (BFS tree), and $\mathcal{MIST}$ is an algorithm that uses the BFS tree to compute an MIS tree of the network.

4.2.1 Algorithm $\mathcal{BFST}$

Let $\mathcal{BFST}$ be a silent self-stabilizing breadth-first search tree algorithm which works for any topology under a weakly fair daemon. That is, starting from an arbitrary configuration, $\mathcal{BFST}$ converges to a terminal configuration where a root r and a breadth-first search tree of the network, rooted at r, is output. Henceforth, we denote by $\texttt{Level}_{\texttt{BFS}}(p)$ the level of any process p in the breadth-first search tree computed by $\mathcal{BFST}$.

Many silent self-stabilizing breadth-first search tree algorithms have been given in the literature. One of the first silent self-stabilizing algorithm for that problem is given in [HC92]. However, it was designed for arbitrary rooted networks. The silent self-stabilizing algorithm for identified networks given in [DLV11a] can be used to implement $\mathcal{BFST}$. Actually, this algorithm is a leader election, but, as most of the existing silent self-stabilizing leader election algorithms, it also builds a BFS tree that is rooted at the elected process. This algorithm stabilizes in $O(n)$ rounds using $O(\log n)$ bits per process, and does not require processes to know any upper bound on the size n or the diameter $\mathcal{D}$ of the network.

4.2.2 Algorithm $\mathcal{MIST}$

Let r be the root of the BFS tree computed by $\mathcal{BFST}$. Let $\prec$ be an order on processes defined as follows : $p \prec q$ if and only if $(\|p, r\|, p)$ is smaller than $(\|q, r\|, q)$ in the lexical ordering of pairs. Using the outputs of $\mathcal{BFST}$, $\mathcal{MIST}$ computes the MIS tree of the network which is lexically first w.r.t. to $\prec$. The formal description of $\mathcal{MIST}$ is given in Algorithm 1. In $\mathcal{MIST}$, the program of each process p contains two variables:

- The Boolean variable $p.dominator$, which determines if p is in the independent set or not.
- The pointer variable $p.parent$, which points to the parent of p in the MIS tree.

Every process p such that $p.dominator = \text{TRUE}$ is said to be a *dominator*, otherwise it is said to be *dominated*. Eventually, the set $\{p \in V : p.dominator\}$ is fixed and forms a maximal independent set of the network thanks to Action `SetDominator`.

To decide its status, dominator or dominated, each process uses a *priority*, noted $\texttt{Priority}(p)$, which is defined by the tuple $(\texttt{Level}_{\texttt{BFS}}(p), p)$ (*n.b.*, $\texttt{Level}_{\texttt{BFS}}(p)$ is eventually equal to the distance of p to the root of the BFS tree). According to the priorities and the status of its neighbors, p decides its status as follows: p is a dominator if and only if each neighbor q is either dominated or satisfies $\texttt{Priority}(q) > \texttt{Priority}(p)$, where $>$ is the strict lexical ordering. According to this rule, the root of the BFS tree is the process of minimum priority and consequently is eventually definitely a dominator. All its neighbors become dominated, and so on. Hence, eventually, the set of dominator processes is a maximal independent set.

Each process must choose a parent such that the parent links form a spanning tree, and the set of processes at even levels is exactly the set of dominators. The root r sets its parent variable to r. All other processes choose as parent the neighbor having a status different of their own of minimum priority. This forces a strict alternation between status dominator/dominating along every path of the tree. As the root is at level zero and of dominating status, this alternation makes the tree an MIS tree.

Algorithm 1 $\mathcal{MIST}$, code for each process p

Input :

$\texttt{Level}_{\texttt{BFS}}(p) \in \mathbb{N}$

Variables:

$p.dominator$: Boolean
$p.parent \in \mathcal{N}(p) \cup \{p\}$

Macros:

$\texttt{Priority}(p) = (\texttt{Level}_{\texttt{BFS}}(p), p)$
$\texttt{Dominator}(p) = \forall q \in \mathcal{N}(p), \neg q.dominator \vee \texttt{Priority}(q) > \texttt{Priority}(p)$
$\texttt{Parent}(p) =$ **if** $\texttt{Level}_{\texttt{BFS}}(p) = 0$
then p
else $q \in \mathcal{N}(p)$:
$\texttt{Priority}(q) = \min\{\texttt{Priority}(q') :$
$q' \in \mathcal{N}(p) \wedge q'.dominator \neq p.dominator\}$

Actions:

(1) `SetDominator` :: $p.dominator \neq \texttt{Dominator}(p) \longrightarrow p.dominator \leftarrow \texttt{Dominator}(p)$
(2) `SetParent` :: $p.parent \neq \texttt{Parent}(p) \longrightarrow p.parent \leftarrow \texttt{Parent}(p)$

4.3 Correctness and Complexity Analysis

According to Corollary 1 on page 37, to show the correctness of $\mathcal{MIST} \circ \mathcal{BFST}$, we show that $\mathcal{MIST}$ constructs an MIS tree starting from any configuration where no action of $\mathcal{BFST}$ is enabled. In such a configuration, a BFS tree T_{BFS} rooted at some process is available. In the following, we denote by r the root of T_{BFS}, which will be also the root of the MIS tree.

The following two lemmas show that $\mathcal{MIST}$ stabilizes in $O(n)$ rounds after $\mathcal{BFST}$ has stabilized.

Lemma 3 *Starting from any configuration where no action of $\mathcal{BFST}$ is enabled, all actions* `SetDominator` *are disabled forever after at most n rounds.*

Proof. Let γ be a configuration where no action of $\mathcal{BFST}$ is enabled. From γ, $\texttt{Priority}(p)$ is fixed forever for every process p. Let $p_1,\ldots,p_n$ the list of processes ordered by $\prec$ (the lexical ordering w.r.t. priorities) in γ. We show the lemma by induction on the rank of every process in the ordering.

- **Base case:** In γ, $p_1 = r$ and $\texttt{Priority}(p_1) = (0, r)$. So, if $p_1.dominator \neq$ TRUE, p_1 is continuously enabled to set $p_1.dominator =$ TRUE. Once, $p_1.dominator =$ TRUE, action `SetDominator` is disabled at p_1 forever. So, after at most one round from γ, action `SetDominator` of p_1 is disabled forever.

- **Inductive Hypothesis:** Let j a positive integer. Assume that for every process p_i such that $i \leq j$, action `SetDominator` is disabled forever at p_i after at most i rounds from γ.

- **Inductive step:** Consider process p_{j+1} in the first configuration of the $(j+1)^{st}$ round from γ. Every neighbor q of p_{j+1} has priority that is fixed forever; moreover if $\texttt{Priority}(q) < \texttt{Priority}(p_{j+1})$, then the value $q.dominator$ is fixed forever by the induction hypothesis. So, either action `SetDominator` is disabled at p_{j+1} or it is continuously enabled. Hence, at the end of the current round, the value of p_{j+1} is fixed forever and the induction holds.

The maximum rank being n, the lemma is verified. □

Lemma 4 *Starting from any configuration where no action of $\mathcal{BFST}$ is enabled, if at least $n+1$ additional rounds have executed, no action of $\mathcal{MIST}$ is enabled.*

Proof. Let γ be a configuration where no action of $\mathcal{BFST}$ is enabled. By Lemma 3, after at most n rounds from γ, no action `SetDominator` is enabled. So, from that point, the values of $\texttt{Priority}(p)$ and $p.dominator$ are fixed forever, for every process p. Now, for all processes, the guard of action `SetParent` only depends on these values. So, after at most one additional rounds, no action of $\mathcal{MIST}$ can ever again be enabled, and we are done. □

We now consider any terminal configuration γ of $\mathcal{MIST} \circ \mathcal{BFST}$. Let I the set of all dominator processes in γ, that is, the set of all processes p such that $p.dominator = \text{TRUE}$ in γ.

The following three technical lemmas are used in order to prove Lemma 8 which states the correctness of $\mathcal{MIST} \circ \mathcal{BFST}$.

Lemma 5 *In any terminal configuration γ of $\mathcal{MIST} \circ \mathcal{BFST}$, I is a maximal independent set of the network.*

Proof. Suppose the set I is not independent, then there exist two neighbors p and q such that $p.dominator$ and $q.dominator$. Then, either $\texttt{Priority}(p) < \texttt{Priority}(q)$ or $\texttt{Priority}(q) < \texttt{Priority}(p)$. In the first case, Action `SetDominator` is enabled at q, in the latter Action `SetDominator` is enabled at p, contradiction.

Suppose the independent set I is not maximal, then there exists a process p such that $\neg p.dominator$ and for every neighbor q of p, $\neg q.dominator$. Then Action `SetDominator` is enabled at p, contradiction. □

In γ, r is the only process such that $\texttt{Level}_{\texttt{BFS}}(r) = 0$. By the definition of $\texttt{Parent}(p)$, we then have:

Remark 1 *In γ, for every process p, either $p = r$ and $p.\text{parent} = r$ or $p \neq r$ and $p.\text{parent} \in \mathcal{N}(p)$.*

Lemma 6 *In any terminal configuration γ of $\mathcal{MIST} \circ \mathcal{BFST}$, for every process $p \neq r$,* $\texttt{Priority}(p.\text{parent}) < \texttt{Priority}(p)$.

Proof. We consider two cases, according to the status of p:

- $p \in I$. Then, by Lemma 5, $\forall q \in \mathcal{N}(p)$, $q.dominator = \text{FALSE}$, in particular for $q = \texttt{Parent}_{\texttt{BFS}}(p)$. Note that $\texttt{Level}_{\texttt{BFS}}(\texttt{Parent}_{\texttt{BFS}}(p)) = \texttt{Level}_{\texttt{BFS}}(p) - 1$. Thus, by definition of the Macro $\texttt{Parent}(p)$, $\texttt{Level}_{\texttt{BFS}}(p.parent) = \texttt{Level}_{\texttt{BFS}}(\texttt{Parent}_{\texttt{BFS}}(p))$. Consequently, $\texttt{Priority}(p.parent) < \texttt{Priority}(p)$.
- $p \notin I$. Then $\neg\texttt{Dominator}(p)$. Now, as no two processes have equal priority, we have $\exists q \in \mathcal{N}(p)$, $\texttt{Priority}(p) > \texttt{Priority}(q) \wedge q.dominator$. So, $\texttt{Priority}(p.parent) \leq \texttt{Priority}(q)$ by definition of Macro $\texttt{Parent}(p)$. Consequently, $\texttt{Priority}(p.parent) < \texttt{Priority}(p)$.

□

In the following, we denote by T_{MIS} the subgraph induced by the values of the parent pointers of $\mathcal{MIST}$ in the terminal configuration γ. Formally, $T_{MIS} = (V, E_{MIS})$, where E_{MIS} is the set $\{\{p, p.parent\} : p \in V \setminus \{r\}\}$ defined in γ. (Recall that r is the unique process such that $r.parent = r$ in γ, by Remark 1.)

Lemma 7 *In any configuration where no action of $\mathcal{MIST} \circ \mathcal{BFST}$ is enabled, T_{MIS} is a spanning tree of the network.*

Proof. We show by contradiction that T_{MIS} is connected and acyclic:

- Suppose T_{MIS} is not acyclic. Then, there exists a elementary cycle in $C = (c_0, c_1, \ldots, c_m = c_0)$ such that $\forall i \in [0..m-1]$, $c_i.parent = c_{i+1}$ and $m > 0$. By Remark 1, $r \notin C$. By Lemma 6, $\forall i \in [0..m-1]$, $\texttt{Priority}(c_i) < \texttt{Priority}(c_{i+1})$. By transitivity, $\texttt{Priority}(c_0) < \texttt{Priority}(c_m)$, that is, $\texttt{Priority}(c_0) < \texttt{Priority}(c_0)$, contradiction.
- Suppose T_{MIS} is not connected, then there exist at least two connected components in T_{MIS}. At least one component, noted G', does not contain the root r. Every process $p \in G'$ has a parent in G', by Macro $\texttt{Parent}(p)$. Hence, there are as many edges as processes in G', *i.e.*, there is a cycle in G'. As T_{MIS} is acyclic, we obtain a contradiction.

□

In the following, we denote by $\texttt{Level}_{\texttt{MIS}}(p)$ the level of any process p in the MIS tree T_{MIS} computed by algorithm $\mathcal{MIST}$.

Lemma 8 *In any configuration where no action of $\mathcal{MIST} \circ \mathcal{BFST}$ is enabled, T_{MIS} is an MIS tree of the network.*

Proof. By Lemma 7, T_{MIS} is a spanning tree of the network. By Lemma 5, I is an MIS of the network. We now show that the even levels of T_{MIS} form I. Formally, we prove that $\texttt{Level}_{\texttt{MIS}}(p)$ is even if and only if $p.dominator$ for all $p \in V$, by induction on $\texttt{Level}_{\texttt{MIS}}(p)$.

First, the root process r is necessarily in I. For the inductive step, let p be a process other than r, and let $L = \texttt{Level}_{\texttt{MIS}}(p) > 0$. By the inductive hypothesis, $\texttt{Level}_{\texttt{MIS}}(q)$ is even if and only if $q.dominator = \text{TRUE}$ for all q such that $\texttt{Level}_{\texttt{MIS}}(q) = L - 1$.

Note that $\texttt{Level}_{\texttt{MIS}}(p.parent) = L - 1$. By Macro $\texttt{Parent}(p)$, $p.parent.dominator \neq p.dominator$. Since L is even if and only if $L - 1$ is not even, we are done. □

We can require that $\mathcal{BFST}$ stabilize in $O(n)$ rounds and use $O(\log n)$ space per process [DLV11a]. So, by Corollary 1 (page 37), Lemmas 4 and 8, we have:

Theorem 7 *$\mathcal{MIST} \circ \mathcal{BFST}$ is a silent self-stabilizing algorithm that builds an MIS tree within $O(n)$ rounds using $O(\log n)$ space per process for any topology under a weakly fair daemon.*

4.4 Height of the MIS Tree

The next property establishes a bound on the height of the MIS tree computed by $\mathcal{MIST} \circ \mathcal{BFST}$. We then illustrate this property with an example matching the bound. To show the property, we need the following technical lemma.

Lemma 9 *In any terminal configuration of $\mathcal{MIST} \circ \mathcal{BFST}$, if p is a non-root process at even level of T_{MIS}, then the process $p.$*parent *is at level $\texttt{Level}_{\texttt{BFS}}(p) - 1$ in T_{BFS}.*

Proof. As p is a dominator process, none of its neighbors is a dominator, by Lemma 5. Since p is not the root, $\texttt{Parent}_{\texttt{BFS}}(p)$ is defined. To sum up, $\texttt{Parent}_{\texttt{BFS}}(p) \in \mathcal{N}(p)$ and $\texttt{Level}_{\texttt{BFS}}(\texttt{Parent}_{\texttt{BFS}}(p)) = \texttt{Level}_{\texttt{BFS}}(p) - 1$, so $\min\{\texttt{Level}_{\texttt{BFS}}(q) : q \in \mathcal{N}(p) \wedge q.dominator \neq p.dominator\} = \texttt{Level}_{\texttt{BFS}}(p) - 1$. By definition, for all q, $\texttt{Level}_{\texttt{BFS}}(q) < \texttt{Level}_{\texttt{BFS}}(p)$ implies $\texttt{Priority}(q) < \texttt{Priority}(p)$. By Macro $\texttt{Parent}(p)$, we are done. □

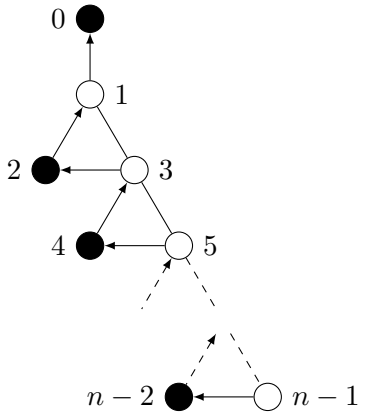


Figure 4.2 – Worst case example for MIS tree height.

Property 2 *In any terminal configuration of $\mathcal{MIST} \circ \mathcal{BFST}$, the height of the computed MIS tree T_{MIS} of G is at most $2 \times \mathcal{D}$, where $\mathcal{D}$ is the diameter of G.*

Proof. Let h be the height of T_{BFS}. Let $\sigma = (p_\ell, p_{\ell-1}, \ldots, p_0 = r)$ be any path in T_{MIS} from a leaf to the root. That is, p_ℓ is a leaf, and $p_j = p_{j+1}.parent$ for all $j < \ell$.

Since T_{MIS} is 2-colored w.r.t. *dominator* variables, any path in T_{MIS} is also 2-colored w.r.t. *dominator* variables. Moreover, $p_0.dominator$ = TRUE, so $p_j.dominator$ = TRUE if and only if j is even, for all $j < \ell$.

Since $\texttt{Priority}(p_{j+1}) > \texttt{Priority}(p_j)$ (Lemma 6), we have:

(a) $\texttt{Level}_{\texttt{BFS}}(p_{j+1}) \geq \texttt{Level}_{\texttt{BFS}}(p_j)$ for all $j < \ell$.

By Lemma 9, $\texttt{Level}_{\texttt{BFS}}(p.parent) < \texttt{Level}_{\texttt{BFS}}(p)$ for any dominator process $p \neq r$. Thus:

(b) $\texttt{Level}_{\texttt{BFS}}(p_{j+1}) > \texttt{Level}_{\texttt{BFS}}(p_j)$ for all odd j.

From (a) and (b), it follows that:

(c) At most two processes of σ can be on any one level of T_{BFS}.

By definition of T_{BFS}:

(d) $p_0 = r$ is the only process of σ at level 0 in T_{BFS}.

By definition of T_{BFS} and (d), p_1 (if defined) is at level 1 in both T_{BFS} and T_{MIS}. Then, by (b), p_2 (if defined) is not at the same level in T_{BFS} as p_1. So, p_0 and p_2 are not at the same level as p_1 in T_{BFS}, that is:

(e) p_1 is the only process of σ at level 1 in T_{BFS}.

Hence, among the $\ell + 1$ processes of σ, there are exactly one process at level zero of T_{BFS}, one process at level 1 of T_{BFS}, and for every other level x of T_{BFS}, there are at most two processes of σ at level x by (c). Hence, $\ell \leq 2 \times (h-1) + 2$, that is, $\ell \leq 2 \times h \leq 2 \times \mathcal{D}$. □

Figure 4.2 exhibits the upper bound on the height of T_{MIS}, depending on the diameter $\mathcal{D}$ of the network. Even processes have the same parent in both T_{BFS} and T_{MIS}, whereas odd ones have their parent in T_{MIS} at the same level in T_{BFS}. It is not possible to increase the height of T_{MIS} more than once per level of T_{BFS}, thus the height of T_{MIS} is at most twice the one of T_{BFS}, that is $2 \times \mathcal{D}$.

4.5 MIS Construction and Nick's Class

From Theorem 7, our algorithm $\mathcal{MIST} \circ \mathcal{BFST}$ builds an MIS tree in $O(n)$ rounds. In this section, we show that finding an algorithm with a sublinear time complexity for computing an MIS tree of a general network could be very hard, and may be impossible.

4.5.1 Nick's Class

Nick's Class ($\mathcal{NC}$) [Coo79] is defined to be the set of all problems that can be solved in parallel in polylogarithmic time with polynomially many processors. Thus, there can be no deterministic polylogarithmic time distributed algorithm for any problem which is not in $\mathcal{NC}$.

Recall that $\mathcal{P}$ is the set of all problems that can be deterministically solved in polynomial time. $\mathcal{NC} \subseteq \mathcal{P}$ because a polylogarithmic time parallel computation with polynomially many processors can be emulated by polynomial-time sequential computation. The question, "Is $\mathcal{NC} = \mathcal{P}$?" is still open and considered to be in the same class of difficulty as the question of whether $\mathcal{P} = \mathcal{NP}$. Most researchers suspect that $\mathcal{NC} \neq \mathcal{P}$, meaning believe there to be tractable problems which are "inherently sequential," and cannot be executed in polylogarithmic time up by using parallelism.

A problem $\mathbb{A} \in \mathcal{P}$ is said to be $\mathcal{P}$-*complete* if, given any problem $\mathbb{B} \in \mathcal{P}$, there is $\mathcal{NC}$-*reduction* of $\mathbb{B}$ to $\mathbb{A}$, *i.e.*, a reduction that can be computed in parallel in polylogarithmic time with polynomially many processors. Thus, $\mathcal{NC} = \mathcal{P}$ if and only if there is any one $\mathcal{P}$-complete problem which is in $\mathcal{NC}$.

Now, if we make the usual assumption that $\mathcal{NC} \neq \mathcal{P}$, then any $\mathcal{P}$-complete problem belongs to $\mathcal{P} \setminus \mathcal{NC}$, meaning that the problem is "inherently sequential." Hence, just as we can justify giving up the search for a polynomial time algorithm for any problem that we can prove to be $\mathcal{NP}$-complete, we can justify giving up the search for a fast parallel algorithm for a problem if we can prove that it is $\mathcal{P}$-complete.

Below, we show that the exact problem solved by our MIS Tree construction is $\mathcal{P}$-complete.

4.5.2 $\mathcal{P}$-Completeness of the LFMIS Problem with a Unique Local Minimum

Given a network $G = (V, E)$, Algorithm $\mathcal{MIST} \circ \mathcal{BFST}$ computes an MIS of G, with respect to the priorities ordering $\prec$ defined in Subsection 4.2. Note that there is a natural lexical ordering on the subsets of V, obtained by writing each subset as a list of processes ordered by $\prec$. The MIS computed by our algorithm comes first in this natural lexical ordering of subsets of V, it is thus the *lexically first* maximal independent set of G.

Let denote by $p_1, \ldots, p_n$ the processes of G, ordered by $\prec$. $\mathcal{MIST} \circ \mathcal{BFST}$ takes advantage of an additional property of priorities: There is a unique local minimum, *i.e.*, for any $i > 1$ there is some $j < i$ such that p_j is a neighbor of p_i (Lemma 6).

The lexically first maximal independent set problem on a graph G is equivalent to finding a lexically first maximal clique in the complementary graph G', shown by Cook [Coo85] to be $\mathcal{P}$-complete.

However, $\mathcal{MIST} \circ \mathcal{BFST}$ solves a restricted version of the LFMIS problem, where the ordering is known to have a unique local minimum, and thus we need to

give separate proof that this version is also $\mathcal{P}$-complete. It consists in exhibiting a method to $\mathcal{NC}$-reduce any instance of the $\mathcal{P}$-complete *Circuit Value* problem to an instance of the LFMIS problem with unique local minimum.

A *Boolean circuit* is a straight line program consisting of finitely many assignments of the form

- $x_i \leftarrow$ TRUE,
- $x_i \leftarrow$ FALSE,
- $x_i \leftarrow x_j \wedge x_k$ with $j, k < i$,
- $x_i \leftarrow x_j \vee x_k$ with $j, k < i$, or
- $x_i \leftarrow \neg x_j$ with $j < i$,

where each variable x_i in the program appears on the left side of exactly one assignment. The conditions $j, k < i$ and $j < i$ ensure acyclicity. (This implies in particular that the right side of the first assignment is a constant TRUE or FALSE). The *Circuit Value* (CV) problem is then defined as evaluating the value of variable x_n in such a program, where n is the maximum index. An example of such a program is given in Figure 4.3a. (The program can be also represented as a Boolean circuit, see in Figure 4.4a the circuit corresponding to the program of Figure 4.3a.)

The CV problem has been shown to be $\mathcal{P}$-complete in [Lad75].

Now, we exhibit a method to $\mathcal{NC}$-reduce any instance of the $\mathcal{P}$-complete CV problem to an equivalent instance of the LFMIS problem with unique local minimum, in order to prove that the LFMIS problem with unique local minimum is $\mathcal{P}$-complete.

First, we show in Lemma 10 that any instance of the CV problem can be expressed in the *paired form* as defined hereafter. Next, in the proof of Theorem 8, we consider an arbitrary instance of the CV problem written in the paired form. We then transform it into an intermediate *reduced form*, from which it is easy to finally obtain an equivalent instance of the LFMIS problem with unique local minimum. Of course, every of these three transformations is shown to be an $\mathcal{NC}$-reduction.

Figure 4.3 gives an example of CV problem of the paired form and its reduced form obtained by the method which starts the proof of Theorem 8. Figure 4.4 represents the same programs as Boolean circuits. In Figure 4.5, we show an equivalent instance of the LFMIS problem with unique local minimum, which results of the transformation at the end of the proof of Theorem 8.

Definition 11 (Paired Form) *A Boolean circuit is said to be of the* paired form*, if and only if the number n of variables is even and for every $i \in [1..n]$:*

- *If i is even, the right side of the i^{th} assignment is the negation of the $(i-1)^{\text{th}}$ assigned variable.*
- *Otherwise, i is odd, the right side of the i^{th} assignment is a constant or the conjunction or disjunction of two prior variables.*

Lemma 10 *Any instance of CV problem can be rewritten into an equivalent Boolean circuit of the paired form, in constant time using a polynomial number of processes in parallel.*

Proof. Consider an instance of CV problem containing n variables. Recall that x_i denotes the i^{th} assigned variable of the program. Here, a, b, c, and d denote new variables. Apply the following transformation on each of the n assignments.

- *If the i^{th} assignment at even rank is not $\neg x_{i-1}$.* Then, we have two cases:
  - $i \neq n$*:* Insert $a \leftarrow \neg x_{i-1}$ and $b \leftarrow \neg x_i$ respectively before and after that assignment.
  - $i = n$*:* We have to ensure that the output of the program remains unchanged. So, insert $a \leftarrow \neg x_{i-1}$ before the i^{th} assignment and insert the assignments $b \leftarrow \neg x_i$, $c \leftarrow b \wedge b$, and $d \leftarrow \neg c$ after the i^{th} assignment. Then, the new output will be $d = \neg c = \neg(b \wedge b) = \neg b = \neg\neg x_i = x_i$.

  So, in both cases the truth value of every variable x_k with $k \in [1..n]$ remains unchanged.

- *If the i^{th} assignment at odd rank is a negation $x_i \leftarrow \neg x_j$ with $j < i$ and $i < n$.* Then, replace the i^{th} assignment by $a \leftarrow x_j \vee x_j$, $b \leftarrow \neg a$ and $x_i \leftarrow b \vee b$. In particular, after the transformation, we have $x_i = b \vee b = \neg a \vee \neg a = \neg(x_j \vee x_j) \vee \neg(x_j \vee x_j) = \neg x_j \vee \neg x_j = \neg x_j$. So, the truth value of every variable x_k with $k \in [1..n]$ remains unchanged.

- *If the n^{th} assignment is at an odd rank.* Then, we should add assignments so that the number of assignments of the new program becomes even. Moreover, we have to ensure that the output of the program remains unchanged. We have two cases:
  - *The assignment is a negation $x_n \leftarrow \neg x_j$ with $j < n$.* So, replace the n^{th} assignment by $a \leftarrow x_j \vee x_j$ and $x_n \leftarrow \neg a$. Then, the output remains unchanged since $x_n = \neg a = \neg(x_j \vee x_j) = \neg x_j$.
  - *The assignment is not a negation.* So, add assignments $a \leftarrow \neg x_n$, $b \leftarrow a \wedge a$, and $c \leftarrow \neg b$ at the end of the program. Then, the new output will be $c = \neg b = \neg(a \wedge a) = \neg a = \neg\neg x_n = x_n$.

  So, in both cases the truth value of every variable x_k with $k \in [1..n]$ remains unchanged.

After the transformation, we obtain a program of the paired form. The value of the last variable of this program is the same as the one of the last variable of the initial program. Finally, note that there are $O(n)$ transformations. Each transformation is independent from each other and can be done in constant time. Thus, the whole program transformation can be done in constant time using a polynomial number of processes in parallel. □

		1 :	$y_1 \leftarrow \text{TRUE}$		
		2 :	$y_2 \leftarrow \neg y_1$		
1 :	$x_1 \leftarrow \text{TRUE}$	3 :	$y_3 \leftarrow \neg y_2$		$x_1 = y_3$
2 :	$x_2 \leftarrow \neg x_1$	4 :	$y_4 \leftarrow \neg y_2 \wedge \neg y_3$		$x_2 = y_4$
3 :	$x_3 \leftarrow x_1 \vee x_2$	5 :	$y_5 \leftarrow \neg y_2 \wedge \neg y_3 \wedge \neg y_4$		$x_3 = y_6$
4 :	$x_4 \leftarrow \neg x_3$	6 :	$y_6 \leftarrow \neg y_2 \wedge \neg y_5$		$x_4 = y_5$
5 :	$x_5 \leftarrow x_2 \wedge x_4$	7 :	$y_7 \leftarrow \neg y_2 \wedge \neg y_4 \wedge \neg y_6$		$x_5 = y_7$
6 :	$x_6 \leftarrow \neg x_5$	8 :	$y_8 \leftarrow \neg y_2 \wedge \neg y_7$		$x_6 = y_8$
	(a)		(b)		(c)

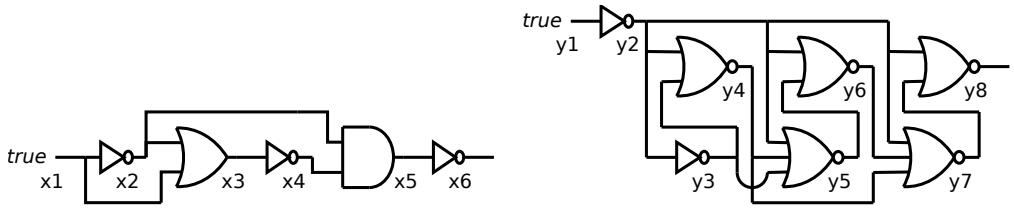
Figure 4.3 – (a) An instance of the CV problem in the paired form, (b) its reduced form, and (c) the correspondence between variables of both instances.

Theorem 8 *The LFMIS problem with unique local minimum is $\mathcal{P}$-complete.*

Proof. Consider an instance of CV problem. Recall that x_i denotes the i^{th} assigned variable of the program. Without loss of generality, we assume that this instance is of the paired form. Indeed, this assumption can be enforced using the $\mathcal{NC}$-reduction given in Lemma 10. Thus, from Definition 11, assuming an even number of variables, we note them $x_1, x_2, \ldots, x_{2n}$. For any $i \in [1..n]$, we will refer to x_{2i-1} and x_{2i} as *partners*. Note that partners always take opposite Boolean values when evaluated.

The rest of the proof is divided into two parts as follows. We first $\mathcal{NC}$-reduce the initial instance of the CV problem into an intermediate *reduced form* (i). Then, we transform that reduced form of the program into an equivalent instance of the LFMIS problem with unique local minimum (ii).

(i) *Reduced Form.* First, we rewrite that program in a reduced form, where variables are noted $y_1, y_2, \ldots, y_{2n+2}$. To begin with, the first assignment will be $y_1 \leftarrow \text{TRUE}$, and the second assignment will be $y_2 \leftarrow \neg y_1$. Then, there will be a one-to-one correspondence between the variables of the initial program and all but the first two variables of the program in the reduced form: For any $i \in [1..n]$, the two variables y_{2i+1} and y_{2i+2} will correspond to the partner variables x_{2i-1} and x_{2i}, in either order. This order will be solved by the rewriting, allowing in particular to know which of y_{2n+1} and y_{2n+2} corresponds to x_{2n}, the output of the initial program. Thus, y_{2i+1} and y_{2i+2} will also have opposite values and we will also refer to these variables as *partners*. We use the following rewriting rules to construct the reduced form of the program, for any $i \in [1..n]$.



(a) (b)

Figure 4.4 – (a) The same instance of the CV problem, and (b) its reduced form, as Boolean circuits.

1. The $(2i+2)^{nd}$ assignment of the reduced program will be $y_{2i+2} \leftarrow \neg y_2 \wedge \neg y_{2i+1}$. That is, y_{2i+2} is assigned to the opposite Boolean value of its odd partner y_{2i+1}, since $\neg y_2 = \text{TRUE}$.

2. The $(2i+1)^{\text{th}}$ assignment of the reduced program will depend on the $(2i-1)^{\text{th}}$ assignment in the initial program:

   (a) If the $(2i-1)^{\text{th}}$ assignment of the initial program is $x_{2i-1} = \text{TRUE}$, then the $(2i+1)^{\text{th}}$ assignment of the reduced program will be $y_{2i+1} \leftarrow \neg y_2$ (that is, TRUE). Thus, y_{2i+1} will correspond to x_{2i-1}, and y_{2i+2} will correspond to x_{2i}.

   (b) If the $(2i-1)^{\text{th}}$ assignment of the initial program is $x_{2i-1} = \text{FALSE}$, then the $(2i+1)^{\text{th}}$ assignment of the reduced program will be $y_{2i+1} \leftarrow \neg y_2$ (that is, TRUE). Thus, y_{2i+1} will correspond to x_{2i}, and y_{2i+2} will correspond to x_{2i-1}.

   (c) If the $(2i-1)^{\text{th}}$ assignment of the initial program is a conjunction $x_{2i-1} \leftarrow x_j \wedge x_k$, let y_p and y_q be the variables corresponding to the *partners* of x_j and x_k, respectively. Then, the $(2i+1)^{\text{th}}$ assignment of the reduced program will be $y_{2i+1} \leftarrow \neg y_2 \wedge \neg y_p \wedge \neg y_q$ (that is, $\text{TRUE} \wedge \neg\neg x_j \wedge \neg\neg x_k = x_j \wedge x_k$). Thus, y_{2i+1} will correspond to x_{2i-1}, and y_{2i+2} will correspond to x_{2i}.

   (d) If the $(2i-1)^{\text{th}}$ assignment of the initial program is a disjunction $x_{2i-1} \leftarrow x_j \vee x_k$, let y_p and y_q be the variables corresponding to x_j and x_k, respectively. Then, the $(2i+1)^{\text{th}}$ assignment of the reduced program will be $y_{2i+1} \leftarrow \neg y_2 \wedge \neg y_p \wedge \neg y_q$ (that is, $\text{TRUE} \wedge \neg(y_p \vee y_q) = \neg(x_j \vee x_k)$). Thus, y_{2i+1} will correspond to x_{2i}, and y_{2i+2} will correspond to x_{2i-1}.

By construction, the partner variables of the reduced program will always be assigned opposite truth values. Through simple induction, we can see that evaluation of the reduced program will assign TRUE to y_1, FALSE to y_2, and to each variable of the reduced program the same value as the corresponding variable in the initial program.

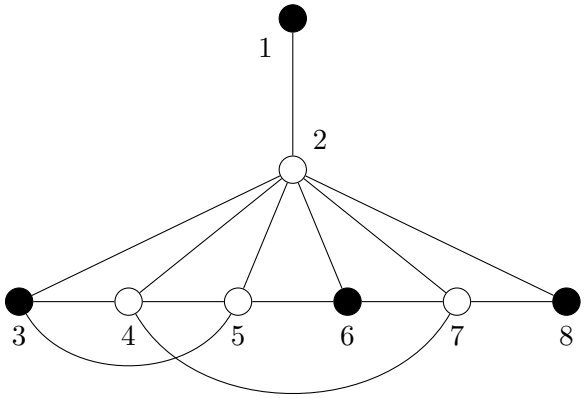


Figure 4.5 – Resulting instance of the LFMIS problem.

(ii) *Equivalent Instance of LFMIS Problem.* Finally, we construct an equivalent instance of the LFMIS problem with unique local minimum as follows. Let G be the network whose ordered (w.r.t. UIDs) list of processes is $p_1, p_2, \ldots, p_{2n+2}$, and where p_1 is the root. For each $1 \leq j < i \leq 2n+2$, p_i is adjacent to p_j if and only if the term $\neg y_j$ appears in the i^{th} assignment of the reduced program. The LFMIS problem with unique local minimum for the reduced program described in Figure 4.3b, and represented as a Boolean circuit in Figure 4.4b, is shown in Figure 4.5. Remark that the distances of every process to p_1 are: $\|p_1, p_1\| = 0$, $\|p_2, p_1\| = 1$, and $\forall 2 < i \leq 2n+2$, $\|p_i, p_1\| = 2$. Consequently, for every $1 < i \leq 2n+2$, $p_{i-1} \prec p_i$.

The first variable y_1 is assigned to TRUE; it is equivalent to having the root process p_1 in the LFMIS. The second variable y_2 is the only one to depend on y_1 and, for every $3 \leq i \leq 2n+2$, y_i depends on y_2; p_2 is the central process of G and the only one at level 1. Every other variable is the conjunction of the negations of some previous variables, which implies that, for every $3 \leq i \leq 2n+2$, local computation of the LFMIS at process p_i only relies on prior processes $p_2, \ldots, p_{i-1}$.

By simple induction on process ordering, we can see that $p_i \in I$ if and only if y_i is assigned the value TRUE in the reduced program, that is, also in the corresponding variable of the initial program.

We note that all the steps of the reduction could be accomplished in parallel in polylogarithmic time with polynomially many processors. Thus, any instance of CV problem can be $\mathcal{NC}$-reduced to an instance of the LFMIS problem with unique local minimum. □

Although the problem is technically open, Theorem 8 justifies not seeking an $O(\mathcal{D})$ time algorithm for computing the LFMIS.

Chapter 5

Labeling in Ordered Trees

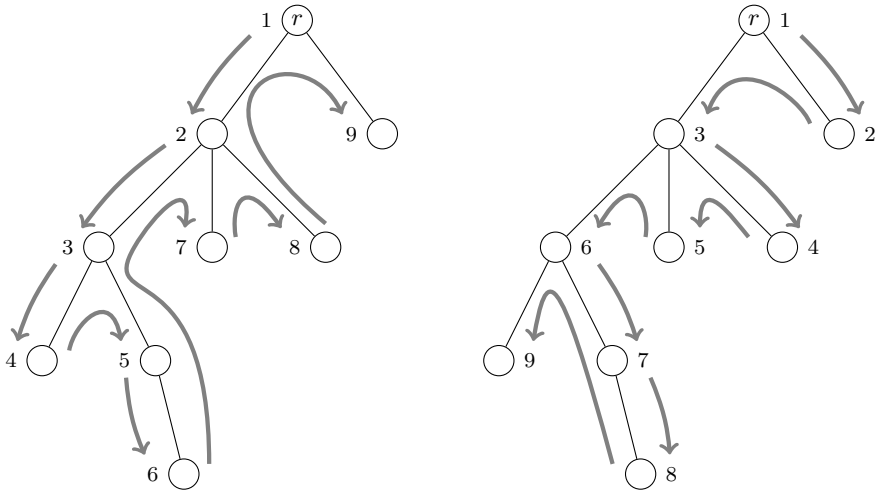
Contents

In this chapter, we present a particular labeling in ordered trees, where a special index, called *guide pair*, is computed for each process. Guide pairs provide a labeling scheme that can be used for various applications [FEP+06], including ordered traversal or navigation in tree networks. We use these labels in Chapter 11 to navigate in the tree network.

For each process p, a *guide pair* is composed of the rank of p in the *preorder* traversal of the ordered tree and the rank of p in the *reverse postorder* traversal of the ordered tree. Figure 5.1 illustrates both traversals of an ordered tree where each process is labeled with its rank in the current traversal. The same ordered tree is shown in Figure 5.2 where each process is labeled with its guide pair.

The notion of guide pairs appeared first in [FEP+06, page 702], where they are used as input of the algorithm. A self-stabilizing algorithm for tree network is given in [CT05b] that computes, for each process, its rank in some traversals of the tree, including the preorder and the reverse postorder traversals. The authors showed that their algorithm stabilizes in $O(n)$ rounds under a central daemon. Note that the algorithm we present in this chapter is an instantiation of the general approach given in [CT05b], however we do not assume a central daemon here. Besides, there exist several self-stabilizing algorithms for other kinds of labeling, *e.g.*, [DGPV01] and [CT05a].

In the following, we first formally define the notion of guide pairs. Then, we describe a self-stabilizing algorithm, called $\mathcal{GUIDE}$ here, to compute guide pairs in tree network, using $O(\delta(p) \log n)$ space per process, where $\delta(p)$ is the degree of p and n the number of processes in the network. Finally, we show that this algorithm is silent and self-stabilizes under a weakly fair daemon, in $O(h)$ rounds, where h is the height of the tree.



(a) Preorder traversal (b) Reverse postorder traversal

Figure 5.1 – (a) Preorder and (b) reverse postorder traversals of an ordered tree.

5.1 Definition of Guide Pairs

Given an ordered tree T, the *guide pair* of a process p in T is the pair of integers i and j such that i and j are, respectively, the rank of p in the *preorder* and *reverse postorder* traversals of T. We denote by $p_1, p_2, \ldots p_m$ the children of the root of T in the left-to-right order. The *preorder traversal* of T is defined, recursively, as follows:

1. Visit the root of T.
2. For each i from 1 to m in increasing order, visit the processes of $T(p_i)$ in *preorder*.

The *reverse postorder traversal* is defined similarly:

1. Visit the root of T.
2. For i from m to 1 in decreasing order, visit the processes of $T(p_i)$ in *reverse postorder*.

If a process p is the i^{th} process of T visited in a preorder traversal of T, we say that the *preorder rank* of p is i. If a process p is the j^{th} process of T visited in a reverse postorder traversal of T, we say that the *reverse postorder rank* of p is j. We note $pre\_ind(p)$ and $post\_ind(p)$ for the *preorder* rank and *reverse postorder* rank of p, respectively. We define the *guide pair* of p to be the ordered pair $guide(p) = (pre\_ind(p), post\_ind(p))$.

We define a partial order on guide pairs: $(i, j) \leq (k, \ell) \Leftrightarrow (i \leq k \wedge j \leq \ell)$.

Remark 2 [Property 1 in [FEP+06]] *If p and q are processes of an ordered tree T, then* guide$(p) \leq$ guide(q) *if and only if p is an ancestor of q.*

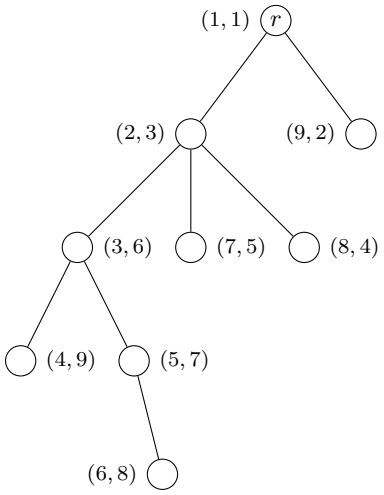


Figure 5.2 – Guide pairs labeling of an ordered tree, the same as in Figure 5.1.

5.2 Algorithm $\mathcal{GUIDE}$

Overview. Consider an ordered tree T rooted at some process r. If we were to compute each guide pair by performing both traversals, like in Figure 5.1, it would take $O(n)$ rounds. We use instead another computation method which converges in sublinear time with respect to the number of processes in the network.

In order to compute the guide pair of any process in T, we have to know the number of predecessors of that process in both preorder and postorder traversals of T. By definition, r is the first process in both traversals, thus its guide pair is always $(1, 1)$. For every non-root process p, we can compute its number of predecessors in both traversals using (1) the guide pair of its parent, *i.e.*, $guide(par(p))$, and (2) for every sibling q of p, the number of processes in the subtree rooted at q, *i.e.*, $|V_{T(q)}|$.

We illustrate this via two examples in Figure 5.4 where the square above each process denotes the count of processes in the subtree rooted at that process. Only counts and guide pairs which take part in the computation are shown, they are actually projected from Figures 5.2 and 5.3.

Preliminaries. Here, we assume that the network is an ordered tree T and the daemon is weakly fair. We also assume that, for every process p in T, for any $q \in \mathcal{N}(p)$, p can determine whether p is the parent of q in T, denoted by $par(q)$. This is implemented for every process q by a variable $q.parent$ such that $q.parent = par(q)$.

Algorithm $\mathcal{GUIDE}$ is actually a hierarchical collateral composition of two algorithms: $\mathcal{GUIDE} = \mathcal{CGP} \circ \mathcal{COUNT}$, where $\mathcal{COUNT}$ computes the number of processes for every subtree of the network in a bottom-up wave, and $\mathcal{CGP}$ (for *Compute Guide Pairs*) computes the guide pairs in a top-down fashion. Both $\mathcal{COUNT}$ and $\mathcal{CGP}$ use $p.parent$ as input in the program of every process p.

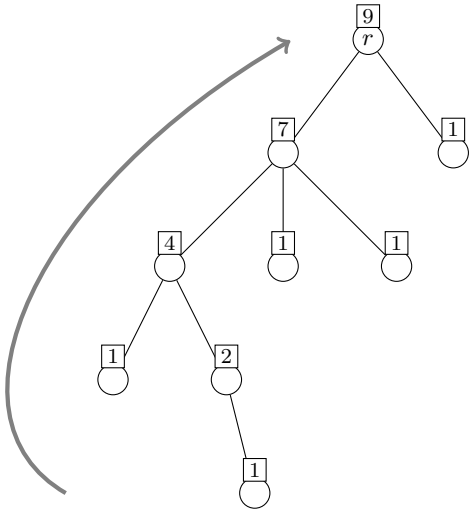


Figure 5.3 – Processes count for each subtree of the same ordered tree as in Figure 5.2.

5.2.1 Algorithm $\mathcal{COUNT}$

$\mathcal{COUNT}$ is implemented as a single bottom-up wave that computes the number of processes in each subtree, as shown in Figure 5.3 where the square above each process denotes the count of processes in the subtree rooted at that process.

The local program of $\mathcal{COUNT}$ for each process is given in Algorithm 2.

Algorithm 2 $\mathcal{COUNT}$, code for each process p

Input :

$p.parent \in \mathcal{N}(p) \cup \{p\}$

Variable:

$p.subcount \in \mathbb{N}$

Macros:

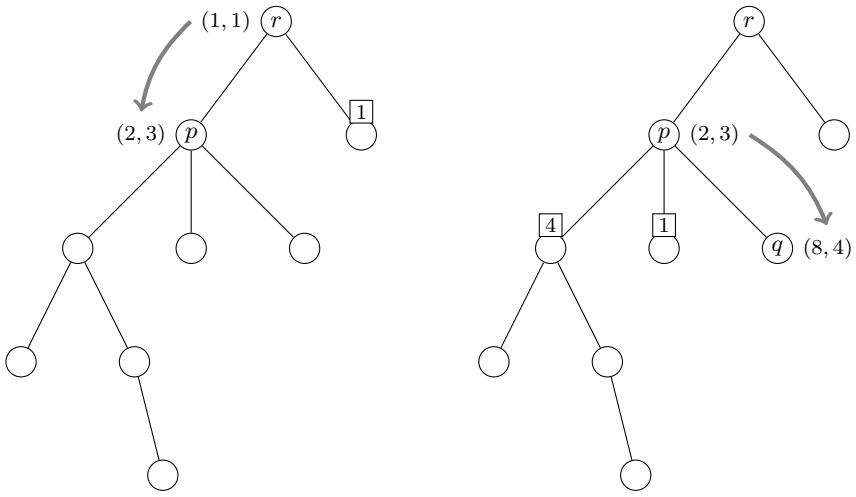
$\texttt{Children}(p) \quad = \quad \{q \in \mathcal{N}(p) : q.parent = p\}$

$\texttt{Subcount}(p) \quad = \quad 1 + \sum_{q \in \texttt{Children}(p)} q.subcount$

Action:

(1) `SetCnt` :: $p.subcount \neq \texttt{Subcount}(p) \longrightarrow p.subcount \leftarrow \texttt{Subcount}(p)$

Recall that $T(p)$ denotes the subtree of T rooted at p. Each process p has only one variable: $p.subcount$ which holds $|V_T(p)|$. Its value is maintained by Action `SetCnt`. The legitimacy predicate of $\mathcal{COUNT}$ is simply that $p.subcount = |V_{T(p)}|$ for all processes p.



(a) Computation of *guide*(p) (b) Computation of *guide*(q)

Figure 5.4 – Computation of guide pairs at processes (a) p and (b) q in an ordered tree, the same as in Figures 5.2 and 5.3.

5.2.2 Algorithm $\mathcal{CGP}$

Overview of $\mathcal{CGP}$. $\mathcal{CGP}$ uses the values of *subcount* computed by $\mathcal{COUNT}$, in order to implement the computation of guide pairs shown in Figure 5.4, but in a *distributed* way.

Consider for example the non-root process q in Figure 5.4b, it cannot read the variables *subcount* of its siblings, since they are not neighbors. However, p, the parent of q can read the variables of its own children which are the siblings of q.

In $\mathcal{CGP}$, each process p evaluates, for each of its children q, the number of predecessors of q both in the *preorder* and *reverse postorder* traversals of the tree T. In order to write and read these values, each process p associates an index number to each of its children with respect to the local left-to-right order $\prec_p$. Reading these values from its parent, each non-root process computes its own guide pair. The guide pair of the root r is set to $(1, 1)$.

Roadmap. We first present the variables of $\mathcal{CGP}$ and their meanings. Then, using these variables, we explain implementation details of the algorithm. We present next the functions and finally the actions of $\mathcal{CGP}$.

Variables of $\mathcal{CGP}$. For each process p, the following variables represent its resulting guide pair:

1. $p.pre\_ind$, $p.post\_ind$, integers, which converge to the *preorder* and *reverse postorder* ranks of p, respectively. Thus, we will denote the guide pair of p by $p.guide = (p.pre\_ind, p.post\_ind)$.

For each process p, the following array variables represent, for each of its children q, the number of predecessors of q in both traversals:

2. $p.child\_pre\_pred[k]$, $p.child\_post\_pred[k]$, integers, defined for all $1 \leq k \leq \delta(p) - 1$.

   For all $1 \leq k \leq \delta(p) - 1$, $p.child\_pre\_pred[k]$ is set to the number of predecessors of the k^{th} child of p in the *preorder* traversal of T; and $p.child\_post\_pred[k]$ is set to the number of predecessors of the k^{th} child of p in the *reverse postorder* traversal of T.

   Hence, each process p computes its guide pair to be

   $$(p.parent.child\_pre\_pred[k] + 1, p.parent.child\_post\_pred[k] + 1)$$

   where k is the index of p in left-to-right order of its parent.

For each process p, the following array variable represents, for each of its children q, the rank of q in the local left-to-right order $\prec_p$:

3. $p.child[k] \in \mathcal{N}(p)$, for all $1 \leq k \leq \delta(p) - 1$. This array is maintained by Action `SetChld`. For all $1 \leq k \leq \delta(p) - 1$, $p.child[k]$ is set to the k^{th} child in p's local ordering of `Children`(p).

   Note that this variable only enables each non-root process q to know its index in the local left-to-right order of its parent p, so to access the appropriate value in the other array variables of p.

Implementation Details of $\mathcal{CGP}$. We now give an intuitive explanation of how $\mathcal{CGP}$ computes the values of $p.pre\_ind$ for all p. The values of $p.post\_ind$ are computed similarly.

Suppose that p is the i^{th} process visited in a *preorder* traversal of T. In this case, i is the correct value of $p.pre\_ind$. $\mathcal{CGP}$ works by computing `NumPreorderPreds(p)`, the number of predecessors of p in the preorder traversal, which is the correct value of $p.pre\_ind - 1$.

First, by definition, $\texttt{NumPreorderPreds}(r) = 0$. Then, for every non-root process p, $\texttt{NumPreorderPreds}(\texttt{p})$ is computed by $p.parent$ and stored in the variable $p.parent.child\_pre\_pred[k]$, where p is the k^{th} child of $p.parent$ in left-to-right order. In order to compute these values for all its children, $p.parent$ must have computed its own value of pre_ind as well as the sizes of all of its subtrees. If $k = 1$, then $\texttt{NumPreorderPreds}(\texttt{p}) = p.parent.pre\_ind$, since $p.parent$ is the immediate predecessor of its leftmost child in the *preorder* visitation. Thus, $p.parent.child\_pre\_pred[1] \leftarrow p.parent.pre\_ind$. $p.parent.child\_pre\_pred[2]$ is obtained by adding the subtree size of the leftmost child of $p.parent$ to $p.parent.child\_pre\_pred[1]$, since all members of that subtree are predecessors of the second child of $p.parent$.

In general, the number of predecessors of p is equal to $p.parent.pre\_ind$ plus the sum of the sizes of the leftmost $k-1$ subtrees of $p.parent$. Similarly, the values of the array $p.parent.child\_post\_pred$ are computed from right to left. p then executes:

$$\begin{aligned} p.pre\_ind &\leftarrow p.parent.child\_pre\_pred[k] + 1 \\ p.post\_ind &\leftarrow p.parent.child\_post\_pred[k] + 1 \end{aligned}$$

Functions of $\mathcal{CGP}$. Using its variables and those of its neighbors, each process p can compute the following functions:

- $\texttt{MyOrder}(p)$. It is only defined for non-root processes. If there exists k, $1 \leq k \leq \delta(p.parent) - 1$, such that $p.parent.child[k] = p$, then $\texttt{MyOrder}(p)$ returns k. Otherwise, the values in $p.parent.child$ have not stabilized yet and $\texttt{MyOrder}(p)$ returns 1.

  Once the system has stabilized, $\texttt{MyOrder}(p)$ returns the index of the non-root process p in the local left-to-right order of its parent.

- $\texttt{ChildIndex}(p, q) = |\{q' \in \texttt{Children}(p) : q' \prec_p q\}| + 1$. $\texttt{ChildIndex}(p, q)$ returns the index of the child q of process p in the local left-to-right order of p.

- $\texttt{EvalChild}(p, k)$ returns the local name of the k^{th} child of p. That is, $\texttt{EvalChild}(p, k)$ returns $q \in \texttt{Children}(p)$ such that $\texttt{ChildIndex}(p, q) = k$.

- $\texttt{EvalChildPrePred}(p, k)$. If $k = 1$, then $\texttt{EvalChildPrePred}(p, k)$ returns $p.pre\_ind$; else $\texttt{EvalChildPrePred}(p, k)$ returns $p.child\_pre\_pred[k-1] + p.child[k-1].subcount$.

  Once the system has stabilized, $\texttt{EvalChildPrePred}(p, k)$ returns the number of predecessors of the k^{th} child of p in the *preorder* traversal of T.

- `EvalChildPostPred`(p, k). If $k = \delta(p) - 1$, then `EvalChildPostPred`(p, k) returns $p.post\_ind$, else `EvalChildPostPred`(p, k) returns $p.child\_post\_pred[k+1] + p.child[k+1].subcount$.

  Once the system has stabilized, `EvalChildPostPred`(p, k) returns the number of predecessors of the k^{th} child of p in the reverse postorder traversal of T.

Actions of $\mathcal{CGP}$. Actions of $\mathcal{CGP}$ are given in Algorithm 3. For every process p, Actions `SetChld`, `SetChldPrePred`, and `SetChldPostPred` respectively compute the values of array variables $p.child$, $p.child\_pre\_pred$, and $p.child\_post\_pred$. These variables are used for the computation of $p.pre\_ind$ and $p.post\_ind$ done by Actions `SetPreInd` and `SetPostInd` which differs from process r to non-root processes.

Algorithm 3 $\mathcal{CGP}$, code for each process p

Actions for every process p:

(1) `SetChld`
:: $\exists k \in [1..\delta(p)-1], p.child[k] \neq$ `EvalChild`(p,k)
$\longrightarrow$ $\forall k \in [1..\delta(p)-1], p.child[k] \leftarrow$ `EvalChild`(p,k)

(2) `SetChldPrePred`
:: $\exists k \in [1..\delta(p)-1], p.child\_pre\_pred[k] \neq$ `EvalChildPrePred`(p,k)
$\longrightarrow$ $\forall k \in [1..\delta(p)-1], p.child\_pre\_pred[k] \leftarrow$ `EvalChildPrePred`(p,k)

(3) `SetChldPostPred`
:: $\exists k \in [1..\delta(p)-1], p.child\_post\_pred[k] \neq$ `EvalChildPostPred`(p,k)
$\longrightarrow$ $\forall k \in [1..\delta(p)-1], p.child\_post\_pred[k] \leftarrow$ `EvalChildPostPred`(p,k)

Actions for the root process r only:

(4) `SetPreInd` :: $r.pre\_ind \neq 1 \longrightarrow r.pre\_ind \leftarrow 1$

(5) `SetPostInd` :: $r.post\_ind \neq 1 \longrightarrow r.post\_ind \leftarrow 1$

Actions for every non-root process p only:

(4) `SetPreInd`
:: $p.pre\_ind \neq 1 + p.parent.child\_pre\_pred[$`MyOrder`$(p)]$
$\longrightarrow$ $p.pre\_ind \leftarrow 1 + p.parent.child\_pre\_pred[$`MyOrder`$(p)]$

(5) `SetPostInd`
:: $p.post\_ind \neq 1 + p.parent.child\_post\_pred[$`MyOrder`$(p)]$
$\longrightarrow$ $p.post\_ind \leftarrow 1 + p.parent.child\_post\_pred[$`MyOrder`$(p)]$

5.2.3 Correctness and Complexity Analysis

Lemma 11 *$\mathcal{COUNT}$ is self-stabilizing and silent, and converges within $h+1$ rounds from an arbitrary initial configuration to a legitimate configuration.*

Proof. By induction on the height of $T(p)$. Within one round, $p.subcount = 1$ if p is a leaf of T. Otherwise, If $T(p)$ has height t, then, by the inductive hypothesis, for every child q of p, $q.subcount = |V_{T(q)}|$ if at least t rounds have elapsed, and thus $\texttt{Subcount}(p) = |V_{T(p)}|$. Within one more round, $p.subcount = |V_{T(p)}|$. □

Theorem 9 *$\mathcal{GUIDE}$ is self-stabilizing and silent, computes the guide pairs of all processes in $O(h)$ rounds from an arbitrary initial configuration, and works under any weakly fair daemon.*

Proof. According to Corollary 1 (page 37) and Lemma 11, to show that $\mathcal{GUIDE}$ is self-stabilizing, it is sufficient to show that $\mathcal{CGP}$ stabilizes from any silent legitimate configuration of $\mathcal{COUNT}$.

In such a configuration, the value of $p.subcount$ is correct for all p. The variables of $\mathcal{CGP}$ are then computed in a top-down wave which takes $O(h)$ rounds. (We can prove this by induction on the height of processes in the tree, similar to the proof for $\mathcal{COUNT}$.) Once a legitimate configuration is reached, no action is enabled.

Finally, the round convergence time of $\mathcal{GUIDE}$ is equal to the round convergence time of $\mathcal{COUNT}$ ($O(h)$ rounds) plus the number of rounds for $\mathcal{CGP}$ to reach a final configuration from any configuration where the values of all $p.subcount$ are correct ($O(h)$ rounds). □

Part III

k-Clustering

Chapter 6

k-Clustering

Contents

In this part, we are interested in self-stabilizing algorithms for constructing a *k-clustering* of any connected network as defined hereafter.

In this chapter, we first introduce some concepts which will be used through Part III. Then, we explain the motivation behind the two approaches developed in Chapters 7 and 8. Finally, we examine related work in both self-stabilizing and non self-stabilizing settings.

6.1 Key Concepts

In this section, we first introduce the definition of a *k-clustering* of a network. Then, we recall the relationship between k-clustering and k-dominating set.

6.1.1 Definition of k-Clustering

We first give the general definition of a clustering. Then, we derive the definition of a k-clustering from it.

Given a connected graph $G = (V, E)$, a *cluster* of G is defined to be a set $C \subseteq V$, together with a designated vertex $\mathit{Clusterhead}(C) \in C$. A *clustering* of G is a partition of V into distinct clusters. The *size* of a clustering is its number of clusters. A *k-clustering* of G is a clustering in which every member of every cluster is within distance k from its clusterhead. Such cluster is called *k-cluster*.

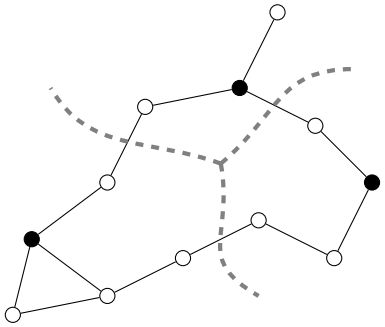


Figure 6.1 – Example of k-clustering with $k = 2$. Clusterheads are black-colored. Borders of clusters are represented by dashed lines.

Notice that there are some homonyms in the literature which are not related to this definition of k-clustering. A partition of a graph (or a space) into k distinct clusters is also referred to as "k-clustering", for example in [Bru78] and [OR00].

Figure 6.1 gives an example of k-clustering of a network with $k = 2$, partitioning it into three 2-clusters.

6.1.2 Relationship with k-Dominating Set

We recall – from page 22 within Section 2.1 – that, given a topology $G = (V, E)$ and a non-negative integer k, a *k-dominating set* of G is a subset of processes D such that every process that is not in D is at distance at most k from a process in D.

Again, please take care of unrelated homonyms in the literature. A k-redundant dominating set, as introduced in [KK03], is a set of processes such that every process of the network either is a member of this set or has at least k members of this set in its neighborhood, that is at distance 1. It is often named " k-dominating set" too, for example in [HLCW07] and [WWTZ12]. Instead, we consider here a set of processes such that every process of the network either is a member of this set or has at least one member of this set at distance at most k from it.

Building a *k-dominating set* in a network is useful because it allows the network to be partitioned into (distinct) k-clusters, that is, it allows to construct a k-clustering of the network, by using every member of the k-dominating set as clusterhead.

The set of clusterheads of a k-clustering is a k-dominating set; conversely, if D is a k-dominating set, a k-clustering is obtained by having every process choose the closest member of D as its clusterhead, ties being resolved arbitrarily.

In Figure 6.1, the set of black-colored processes is also a 2-dominating set of the network.

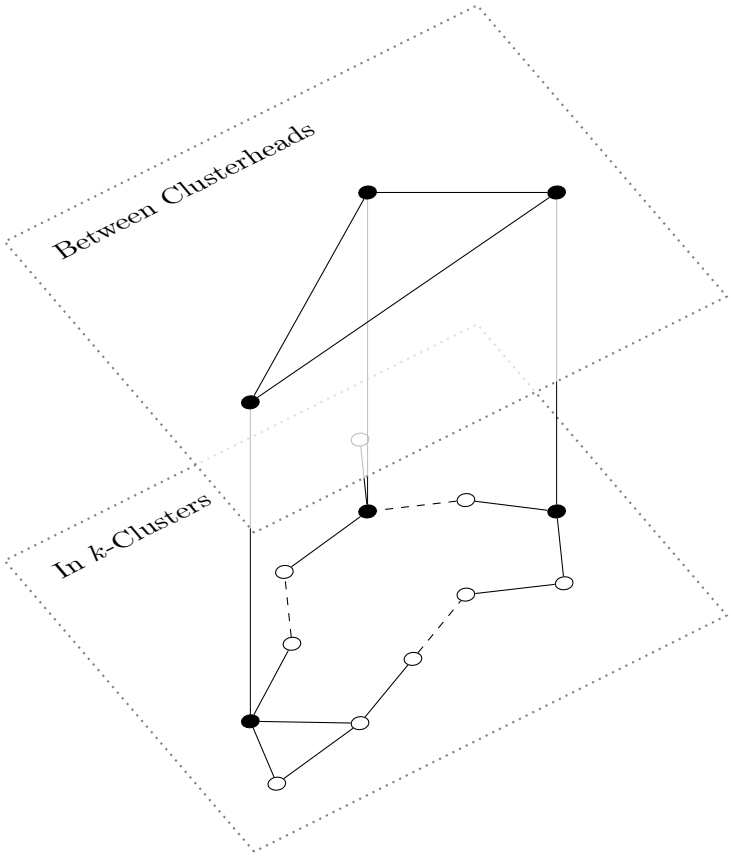


Figure 6.2 – Illustration of routing scheme over the k-clustering of Figure 6.1 with $k = 2$. Edges which are not in a k-cluster are represented by dashed lines.

6.2 Seeking Optimization

A major application of *k-clustering* resides in the implementation of an efficient routing scheme in a network of processes. Indeed, we could rule that a process that is not a clusterhead communicates only with processes in its own k-cluster, and that clusterheads communicate with each other *via* virtual "super-edges," implemented as follows. Given a k-clustering of $G = (V, E)$, two k-clusters C_1 and C_2 are said to be *neighbors* if and only if there exist two processes p and q in V such that $p \in C_1$, $q \in C_2$ and $\{p, q\} \in E$. For every pair of neighboring k-clusters, their respective clusterheads are at distance at most $2k + 1$ from each other. Therefore, every virtual "super-edge" can be implemented as path of length at most $2k + 1$ in the network. An example of this routing scheme is exhibited in Figure 6.2.

Ideally, we would like to find a k-clustering with the minimum number of k-clusters. Respectively, we would like to find a k-dominating set with the minimum number of members. However, these problems are known to be $\mathcal{NP}$-hard [GJ79].

Therefore, we propose to study the two other approaches instead, introduced in Subsections 6.2.1 and 6.2.2, and broadened in Chapters 7 and 8, respectively.

6.2.1 Minimal and Size-Bounded k-Clustering

We first consider the problem of finding a *minimal* k-dominating set, *i.e.*, a k-dominating set which has no k-dominating proper subset. We remark that minimality does not guarantee that a k-dominating set is small. See, for example, Figure 6.3. The singleton $\{v_0\}$ is a minimum thus minimal 1-dominating set. However, the set of black processes is also a minimal 1-dominating set, still it is very large, its size is $n-1$.

In Chapter 7, we address this problem by giving a self-stabilizing algorithm that builds a minimal k-dominating set whose size is bounded by $\left\lceil \frac{n}{k+1} \right\rceil$, where n is the number of processes in the network.

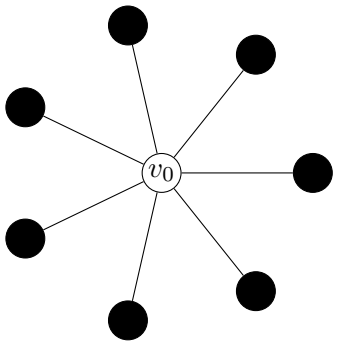


Figure 6.3 – Examples of minimal 1-dominating set.

6.2.2 Competitiveness

We consider here the problem of finding a k-clustering whose size approximates [DS97] the size of a k-clustering of smallest possible cardinality. A k-clustering of a network is α-*competitive* [FM02] if and only if its size is at most α times the size of a minimum k-clustering of this network. By extension, we derive from this definition the notion of α-*competitive* k-dominating set

In Chapter 8, we give a self-stabilizing algorithm to construct a k-clustering of any connected network. We study the case of unit-disk graphs and approximate disk graphs, previously defined in Chapter 2, page 22. These topologies are commonly used to model wireless ad hoc networks. We prove, from a theorem in geometry, that our algorithm builds a competitive k-clustering of such networks.

6.3 Related Work

There are several known self-stabilizing distributed algorithms for finding a k-clustering of an asynchronous network, *e.g.*, [CDDL10, DLV09, DDL09]. The solution in [DLV09] self-stabilizes in $O(k)$ rounds using $O(k \log n)$ space per process. The algorithm given in [CDDL10] self-stabilizes in $O(k.n)$ rounds using $O(k \log n)$ space per process. The algorithm given in [DDL09] self-stabilizes in $O(n)$ rounds using $O(\log k + \log n)$ space per process. All these algorithms work under an unfair daemon. Recall that the set of clusterheads for any k-clustering of a network is a k-dominating set of the network. The k-dominating set computed by the algorithm given in [DDL09] is also minimal. In the same paper, it is shown that every minimal k-dominating set contains at most $\max(1, n/\lceil \frac{k+1}{2} \rceil)$ processes. All the aforementioned self-stabilizing algorithms are written in the locally shared memory model. However, none of them guarantees to output a *competitive* k-clustering or k-dominating set. There are several self-stabilizing solutions that compute a minimal 1-dominating set, *e.g.*, [SRR95, IKK02]. However, the solution for 1-dominating sets does not scale up well to k-dominating sets. In particular it does not maintain interesting bounds on the size of the computed dominating set.

There are several *non self-stabilizing* distributed solutions for finding a k-clustering of a network [APHV00, FM02, SGLA04, Rav05]. Of those, only [FM02] deals with competitiveness. Moreover, they are all written in message-passing model. Deterministic solutions given in [APHV00, FM02] are designed for *asynchronous mobile ad hoc* networks, *i.e.*, they assume networks with a UDG topology. The time and space complexities of the solution in [APHV00] are $O(k)$ and $O(k \log n)$, respectively. Fernandess and Malkhi [FM02] give a k-clustering algorithm that takes $O(n)$ steps using $O(\log n)$ memory per process, provided a BFS tree of the network is already given. In the special case that the network is a UDG, their algorithm is $8k + O(1)$-*competitive*.Actually, in [FM02], a k-cluster is defined to have diameter at most k, while our definition uses radius k. They give competitiveness $4k + O(1)$, which is equivalent to competitiveness $8k + O(1)$ using our definition of k-cluster. Spohn and Garcia-Luna-Aceves [SGLA04] give a distributed solution to a more generalized version of the k-clustering problem. In this version, a parameter m is given, and each process must be a member of m different k-clusters. The time and space complexities of this algorithm for asynchronous networks are not given. Ravelomanana [Rav05] gives a randomized algorithm for synchronous UDG networks whose time complexity is $O(\mathcal{D})$ rounds, where $\mathcal{D}$ is the diameter of the network. In [PU89], the authors consider the problem of deterministically finding a k-dominating set of at most $\left\lceil \frac{n}{k+1} \right\rceil$ processes. Their solution assumes a synchronous system and has $O(k \log^* n)$ time complexity. However, the authors missed one special case, which unfortunately invalidates their proof for some networks. The same flaw is present in some subsequent papers [KP95a, PB04]. Ravelomanana [Rav05] gives a randomized algorithm designed for synchronous UDG networks whose time complexity is $O(\mathcal{D})$ rounds.

6.4 Roadmap of Part III

In this part, we present two approaches, previously discussed in Section 6.2, as follows. We first present a silent self-stabilizing algorithm for constructing a minimal k-dominating set of the network in Chapter 7. Then, in Chapter 8, we give a silent self-stabilizing algorithm for building a k-clustering of the network. We also prove that it builds a competitive k-clustering in some geometric graphs which model wireless ad hoc networks. Finally, we evaluate in simulations the size of the k-clustering (or k-dominating set) computed by our algorithms in Chapter 9.

Chapter 7

Small Minimal k-Dominating Sets

Contents

In this chapter, we give a silent self-stabilizing algorithm for finding a minimal k-dominating set of at most $\lceil \frac{n}{k+1} \rceil$ processes in any connected network.

We first consider the upper bound on the size of minimum k-dominating sets given in [PU89]. We show that the proof given in [PU89] missed a case, and give a correction that does not change the bound.

Next, we give a silent self-stabilizing algorithm, called $\mathcal{SMDS}(k)$, for finding a minimal k-dominating set of small size. To simplify the design of our algorithm, we define it to be a composition of three layers. The first two layers together compute a k-dominating set of at most $\lceil \frac{n}{k+1} \rceil$ processes. The resulting k-dominating set may not be minimal. We apply the algorithm given in [DDL09] in the last layer to remove processes from D until we obtain a minimal k-dominating set. The three layers composed algorithm is proven assuming a weakly fair daemon. The solution stabilizes in $O(n)$ rounds using $O(\log k + \log n + k \log \frac{N}{k})$ bits per process, where N is an upper bound on n, the size of the network. Note that N is introduced to the sole purpose of bounding the space complexity of our solution. Thus, we assume that processes know the value of N, without needing to know the value of n.

Roadmap. In Section 7.1, we give a counterexample for the proof of the upper bound given in [PU89], as well as a correction. Then, our silent self-stabilizing algorithm $\mathcal{SMDS}(k)$ is presented and proven in Section 7.2.

7.1 Upper Bound

In this section, we present an upper bound on the size of the minimum k-dominating set in any connected network. This upper bound originally appeared in [PU89]. However, the proof given in [PU89] overlooked a special case. The same case was overlooked in some other subsequent papers as well [KP95a, PB04].

Below, we exhibit a counterexample to show the special case where the proof of [PU89] is not valid. We then show how to fix the problem without affecting the upper bound.

Let T be an arbitrary spanning tree of $G = (V, E)$ rooted at some process r, that is, any connected graph $T = (V_T, E_T)$ such that $V_T = V$, $E_T \subseteq E$, and $|E_T| = |V_T| - 1$, where the process r is distinguished. In T, the *level* of process p, $lvl_T(p)$, denotes its distance to the root r. The height, $h(T)$, of the tree T, is defined to be $\max_{p \in V_T} lvl_T(p)$, and denoted simply h if T is understood. We write $h(T(p))$ for the height of the subtree $T(p)$ of T rooted at p.

The original proof consists of partitioning the processes of V into sets $T_0, \ldots, T_h$, where $T_i = \{p : lvl_T(p) = i\}$. These sets are merged into $k+1$ disjoint sets $D_0, \ldots, D_k$, where $D_i = \bigcup_{j \geq 0} T_{i+j(k+1)}$.

When $k < h$, the proof in [PU89] claims that (1) the size of the smallest set D_i is at most $\lceil \frac{n}{k+1} \rceil$, and (2) every D_i ($i \in [0..k]$) is k-dominating. The upper bound is then obtained by considering the set D_i of smallest size.

Actually, this latter set is not always k-dominating. The problem arises near the root, where there could be a process which has no ancestor in D_i. For example, consider the case $k = 2$ in the tree network of Figure 7.1. Clearly, D_2 is not a 2-dominating set, because u is not 2-dominated by any process in D_2; $\|u, w\| = 3$.

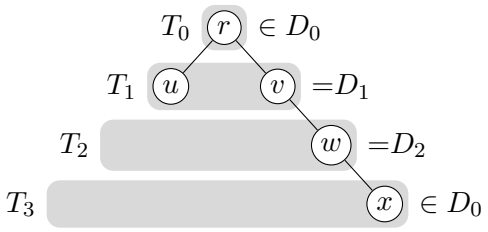


Figure 7.1 – Counterexample of the original proof.

This mistake can be corrected without changing the bound. Actually, the mistake only appears when the smallest D_i ($i \in [0..k]$), say D_ℓ, is not D_0. In this case, a leaf process whose level is strictly less than ℓ may be not k-dominated by any process in D_ℓ (as in the previous example). To correct this mistake we simply proceed as follows. When $k \geq h$ (in this case $|D_0| = 1$) or every D_i ($i \in [0..k]$) has the same size (*i.e.*, $\lceil \frac{n}{k+1} \rceil$), then we choose $D = D_0$. Otherwise, the size of the smallest D_i ($i \in [0..k]$), say $D_{\min}$, is strictly less than $\lceil \frac{n}{k+1} \rceil$ and we choose $D = D_{\min} \cup \{r\}$. In both cases, D is a k-dominating set of size at most $\lceil \frac{n}{k+1} \rceil$.

Theorem 10 *For every connected network $G = (V, E)$ of n processes and for every $k \geq 1$, there exists a k-dominating set D such that $|D| \leq \lceil \frac{n}{k+1} \rceil$.*

Proof. If $n = 0$, then $\lceil \frac{n}{k+1} \rceil = 0 = |\emptyset|$ and $\emptyset$ is a k-dominating set.

Assume now that $n > 0$. Let T be any rooted spanning tree of G rooted at some process r and let $D_0, \ldots, D_k$ be the $k+1$ previously defined sets.

- Assume that $k \geq h$. Then, D_0 only contains r, and every other process is within distance k of r. So, D_0 is a k-dominating set of size $1 \leq \lceil \frac{n}{k+1} \rceil$.
- Assume that $k < h$. Then, for every $i \in [0..k]$, $|D_i| > 0$.
  1. Assume that $\forall i \in [0..k-1]$, $|D_i| = |D_{i+1}|$. Then, $\forall i \in [0..k]$, $|D_i| = \lceil \frac{n}{k+1} \rceil$. Let $v \notin D_0$. The level of v, $lvl_T(v)$, satisfies $lvl_T(v) = x(k+1) + y$, where $x \geq 0$ and $0 < y \leq k$. Let u be the ancestor of v such that $lvl_T(u) = lvl_T(v) - y$ (such an ancestor exists because $y \leq lvl_T(v)$). By definition, $u \in D_0$ and $\|u, v\| \leq k$. Hence, D_0 is a k-dominating set such that $|D_0| = \lceil \frac{n}{k+1} \rceil$.
  2. Assume that there exists $i \in [0..k-1]$ such that $|D_i| \neq |D_{i+1}|$. Let $\min \in [0..k]$ such that $\forall i \in [0..k]$, $|D_{\min}| \leq |D_i|$. Then, $|D_{\min}| < \lceil \frac{n}{k+1} \rceil$. Let $D = D_{\min} \cup \{r\}$. Then, $|D| \leq \lceil \frac{n}{k+1} \rceil$. Let $v \notin D$.
     (a) If $lvl_T(v) \leq k$, then v is at distance at most k from r and $r \in D$.
     (b) If $lvl_T(v) > k$, then $lvl_T(v) = x(k+1) + y$ with $x > 0$, $0 \leq y \leq k$, and $y \neq \min$. If $y > \min$, then let u be the ancestor of v such that $lvl_T(u) = x(k+1) + \min$. If $y < \min$, let u be the ancestor of v such that $lvl_T(u) = (x-1)(k+1) + \min$. By definition, $u \in D$ (more precisely, $u \in D_{\min}$) and $\|u, v\| \leq k$.

     Hence, D is a k-dominating set, and $|D| \leq \lceil \frac{n}{k+1} \rceil$. □

7.2 Algorithm $\mathcal{SMDS}(k)$

In this section, we present a silent self-stabilizing algorithm, called $\mathcal{SMDS}(k)$ (for *Small Minimal k-Dominating Set*), which builds a minimal k-dominating set of at most $\lceil \frac{n}{k+1} \rceil$ processes in any identified network, assuming a weakly fair daemon. This algorithm is a hierarchical collateral composition of three silent self-stabilizing algorithms, $\mathcal{SMDS}(k) = \mathcal{MIN}(k) \circ \mathcal{DS}(k) \circ \mathcal{ST}$, where:

- $\mathcal{ST}$ builds a rooted spanning tree.
- $\mathcal{DS}(k)$ computes a k-dominating set of at most $\lceil \frac{n}{k+1} \rceil$ processes, using the spanning tree built by $\mathcal{ST}$.
- $\mathcal{MIN}(k)$ reduces the k-dominating set built by $\mathcal{DS}(k)$ to a minimal k-dominating set by deleting processes.

We give more details about the three layers of $\mathcal{SMDS}(k)$ in Subsections 7.2.1 to 7.2.3. The complexity of $\mathcal{SMDS}(k)$ is analyzed in Subsection 7.2.4.

7.2.1 Algorithm $\mathcal{ST}$

$\mathcal{ST}$ is any silent self-stabilizing spanning tree algorithm for arbitrary identified networks which works under the weakly fair daemon. The spanning tree built by $\mathcal{ST}$ is *rooted*, meaning that some process of the tree is distinguished as the *root*. For each other process p, let the *parent* of p, be the neighbor of p on the unique shortest path, through the tree, from p to the root. We assume that the output of $\mathcal{ST}$ is the macro $\texttt{Parent}(p)$, which is defined for all processes p. $\texttt{Parent}(p)$ returns $\perp$ if p believes to be the root of the spanning tree, otherwise $\texttt{Parent}(p)$ returns the parent of p. $\mathcal{ST}$ stabilizes w.r.t. the predicate $Spec_{\mathcal{ST}}$ defined as follows: $Spec_{\mathcal{ST}}$ holds if and only if the configuration is terminal, there exists a unique process r such that $\texttt{Parent}(r) = \perp$, and the graph $T = (V, E_T)$ where $E_T = \{\{p, \texttt{Parent}(p)\}, \forall p \in V \setminus \{r\}\}$ is a spanning tree.

The silent self-stabilizing algorithm for identified networks given in [DLV11b] can be used to implement $\mathcal{ST}$. Actually, this algorithm elects a leader; however, as with most existing silent self-stabilizing leader election algorithms, it also builds a spanning tree that is rooted at the leader process. This algorithm stabilizes in $O(n)$ rounds using $O(\log n)$ bits per process, and does not require processes to know any upper bound on the size n or the diameter $\mathcal{D}$ of the network.

From [DLV11b], we have:

Lemma 12 *$\mathcal{ST}$ is a silent algorithm which stabilizes with respect to $Spec_{\mathcal{ST}}$ under a weakly fair daemon.*

7.2.2 Algorithm $\mathcal{DS}(k)$

The formal description of $\mathcal{DS}(k)$ is given in Algorithm 4, page 86. $\mathcal{DS}(k)$ is also silent and uses the spanning tree T built by $\mathcal{ST}$ to compute a k-dominating set of at most $\lceil \frac{n}{k+1} \rceil$ processes. It is based on the construction given in the proof of Theorem 10. Informally, $\mathcal{DS}(k)$ uses the following three variables at each process p:

- $p.color \in [0..k]$. Using this variable, p computes $lvl_T(p)$ mod $(k+1)$ (that is its level in T modulo $k+1$) in top-down fashion using Action FixColor. Hence, once $\mathcal{DS}(k)$ has stabilized, each set D_i, defined in Section 7.1, corresponds to the set $\{p \in V : p.color = i\}$.

- *The integer array $p.pop[i]$ is defined for all $i \in [0..k]$.* In each cell $p.pop[i]$, p computes the number of processes in its subtree $T(p)$ having color i, that is, processes q such that $q.color = i$. This computation is performed in a bottom-up fashion using Action FixPop. Hence, once $\mathcal{DS}(k)$ has stabilized, r knows the size of each set D_i.

- $p.min \in [0..k]$. In this variable, p computes the smallest index of the smallest non-empty set D_i, that is, the least used value to color some processes of the network. This value is evaluated in a top-down fashion using Action FixMin based on the values computed in the array $r.pop$. Once the values of $r.pop$ are correct, the root r can compute in $r.min$ the least used color (in case of equality, the smallest index is chosen). Then, the value of $r.min$ is broadcast down in the tree.

According to Theorem 10, after $\mathcal{DS}(k)$ has stabilized, the set of processes p such that $p = r$ or $p.color = p.min$, *i.e.*, the set $\{p \in V : IsDominator(p)\}$, is a k-dominating set of at most $\lceil \frac{n}{k+1} \rceil$ processes. So, $\mathcal{DS}(k) \circ \mathcal{ST}$ stabilizes w.r.t. the predicate $Spec_{\mathcal{DS}(k)}$ defined as follows: $Spec_{\mathcal{DS}(k)}$ holds if and only if the configuration is terminal and the set $\{p \in V : IsDominator(p) = \text{TRUE}\}$ is a k-dominating set of at most $\lceil \frac{n}{k+1} \rceil$ processes.

We now show the correctness of $\mathcal{DS}(k)$. In the following proofs, we always assume the system starts from a configuration where no action of $\mathcal{ST}$ is enabled. Since $\mathcal{DS}(k)$ does not write into the variables of $\mathcal{ST}$, all variables of $\mathcal{ST}$ are fixed forever in such a configuration. Moreover, a spanning tree is well-defined (using the input $\texttt{Parent}(p)$ of every process p) by Lemma 12. We denote this spanning tree by T and its root by r.

Lemma 13 *Starting from any configuration where no action of $\mathcal{ST}$ is enabled, the variable $p.color$ of every process p is set forever to $lvl_T(p)$ mod $(k+1)$ in at most n rounds.*

Proof. First, remark that:

(a) For every process p, Action FixColor, which has highest priority, whose guard is $\neg ColorOK(p)$, is the only action of p that modifies $p.color$.

We show the lemma by induction on the level of the processes in T.
Let γ be a configuration where no action of $\mathcal{ST}$ is enabled.

- **Base Case:** Let consider the root r (the only process of level 0).

  (b) Predicate $ColorOK(r)$ only depends on the variable $r.color$ and input the $\texttt{Parent}(r)$, which is set forever to $\perp$ at γ.

  Assume that $ColorOK(r)$ holds in γ. Then, $r.color = 0$. Moreover, by (a) and (b), $ColorOK(r)$ holds forever and, consequently, $r.color = 0$ holds forever.

  Assume that $ColorOK(r)$ does not hold in γ. Then, by (a) and (b), Action FixColor is continuously enabled at r. As the daemon is weakly fair, Action FixColor is executed by r in at most one round. Hence, after at most one round from γ, $ColorOK(r)$ becomes TRUE and we reduce to the previous case.

- **Inductive Hypothesis:** Let $j \in \mathbb{N}^*$. Assume that, for every process p such that $lvl_T(p) < j$, the variable $p.color$ is set forever to $lvl_T(p) \bmod (k+1)$ after at most $lvl_T(p) + 1$ rounds from γ.

- **Inductive Step:** Consider any process p such that $lvl_T(p) = j$.

  (c) The predicate $ColorOK(p)$ depends only on the variable $p.color$, input $\texttt{Parent}(p)$ which is fixed to some value in $\mathcal{N}(p)$ from γ, and $\texttt{Parent}(p).color$ which is set forever to $lvl_T(\texttt{Parent}(p)) \bmod (k+1)$ after at most $lvl_T(p)$ rounds from γ by the inductive hypothesis.

  Assume that $ColorOK(p)$ holds after $lvl_T(p)$ rounds from γ. Then, $p.color = (\texttt{Parent}(p).color+1) \bmod (k+1) = (lvl_T(\texttt{Parent}(p)) \bmod (k+1)+1) \bmod (k+1) = lvl_T(p) \bmod (k+1)$. Moreover, by (a) and (c), $ColorOK(p)$ holds forever and, consequently, $p.color = lvl_T(p) \bmod (k+1)$ holds forever.

  Assume that $ColorOK(p)$ does not hold after $lvl_T(p)$ rounds from γ. Then by (a) and (c), Action FixColor is continuously enabled at p from γ. As the daemon is weakly fair, Action FixColor is executed by p in at most one additional round. Hence, in at most $lvl_T(p) + 1$ rounds from γ, $ColorOK(p)$, becomes TRUE and we reduce to the previous case.

As the height of T is bounded by $n-1$, the lemma holds. □

Lemma 14 *Starting from any configuration where:*

- *no action of $\mathcal{ST}$ is enabled, and*
- *the variable $q.color$ of every process q is set forever to $lvl_T(q) \bmod (k+1)$,*

for every process p and every index $i \in [0..k]$, the variable $p.pop[i]$ is set forever to $|\{q \in T(p) : q.color = i\}|$ in at most n rounds.

Proof. First, we remark that:

(a) For every process p, Action FixPop, whose guard is $\neg PopOK(p)$, is the only action of p that modifies $p.pop$.

Let γ be a configuration where:

- no action of $\mathcal{ST}$ is enabled, and
- the variable $q.color$ of every process q is set forever to $lvl_T(q) \bmod (k+1)$.

We also remark that:

(b) From γ, for every process p, $ColorOK(p)$ holds forever – thus forever disabling Action FixColor at p – and, consequently, Action FixPop is enabled at p if and only if $\neg PopOK(p)$ holds.

We now show the lemma by induction on the height of $T(p)$.

- **Base Case:** Consider any process p such that $h(T(p)) = 0$ (p is a leaf process).

  (c) Predicate $PopOK(p)$ depends only on the variables $p.pop$ and $p.color$, the latter being set forever to $lvl_T(p)$ mod $(k+1)$ starting from γ.

  Assume that $PopOK(p)$ holds in γ. Then, $\forall i \in [0..k]$, $p.pop[i] = \texttt{SelfPop}(p,i) = |\{q \in T(p) : q.color = i\}|$. Moreover, by (a)-(c), $PopOK(p)$ holds forever and, and consequently, $\forall i \in [0..k]$, $p.pop[i] = |\{q \in T(p) : q.color = i\}|$ holds forever.

  Assume that $PopOK(p)$ does not hold in γ. Then by (a)-(c), Action FixPop is continuously enabled. As the daemon is weakly fair, Action FixPop is executed by p in at most one round from γ. Then, $PopOK(p)$ becomes TRUE, and we reduce to the previous case.

- **Inductive Hypothesis:** Let $j \in \mathbb{N}^*$. Assume that for every process p such that $h(T(p)) < j$ and every index $i \in [0..k]$, the variable $p.pop[i]$ is set to $|\{q \in T(p) : q.color = i\}|$ after at most $h(T(p)) + 1$ rounds from γ.

- **Inductive Step:** Consider any process p such that $h(T(p)) = j$.

  (d) The predicate $PopOK(p)$ depends only on variables $p.pop$, $p.color$ (which is fixed by assumption), and $q.pop$ of every child q of p in T; these latter variables are fixed after $h(T(p))$ rounds from γ, by the inductive hypothesis.

  Assume that $PopOK(p)$ holds after $h(T(p))$ rounds from γ. Then, $\forall i \in [0..k]$, $p.pop[i] = \texttt{EvalPop}(p,i)$, *i.e.*, $p.pop[i] = (\ \texttt{SelfPop}(p,i) + \sum_{q \in \texttt{Children}(p)} |\{q' \in T(q) : q'.color = i\}|\ ) = |\{q \in T(p) : q.color = i\}|$, by the inductive hypothesis. Moreover, by (a), (b), and (d), $PopOK(p)$ holds forever and, consequently, $\forall i \in [0..k], p.pop[i] = |\{q \in T(p) : q.color = i\}|$ holds forever.

  Assume that $PopOK(p)$ does not hold after $h(T(p))$ rounds from γ. Then, by (a), (b), and (d), Action FixPop is continuously enabled at p. As the daemon is assumed to be weakly fair, p executes Action FixPop in at most 1 round. Hence, in at most $h(T(p)) + 1$ rounds, $PopOK(p)$ becomes TRUE, and we reduce to the previous case.

As the height of T is bounded by $n-1$, the lemma holds. □

The proof of the next lemma follows the same scheme as that of Lemma 13.

Lemma 15 *Starting from any configuration where:*

- *no action of $\mathcal{ST}$ is enabled,*
- *the variable $p.color$ of every process p is set forever to $lvl_T(p) \bmod (k+1)$, and*
- *for every process p and every index $i \in [0..k]$, the variable $p.pop[i]$ is set forever to $|\{q \in T(p) : p.color = i\}|$,*

in at most n rounds, the variable $p.min$ of every process p is set forever to the smallest index $i_{min} \in [0..k]$ that satisfies $|C_{i_{min}}| = \min_{j \in [0..k] \,:\, C_j \neq \emptyset} |C_j|$, where $C_j = \{q \in T : q.color = j\}$ for every $j \in [0..k]$.

From Lemmas 13 to 15, we obtain the following theorem:

Theorem 11 *Starting from any configuration where no action of $\mathcal{ST}$ is enabled, $\mathcal{DS}(k) \circ \mathcal{ST}$ converges in at most $3n$ rounds to a terminal configuration where, for every process p:*

(a) *$p.color = lvl_T(p) \bmod (k+1)$, and*

(b) *$p.min = i_{min}$ where i_{min} is the smallest index in $[0..k]$ that satisfies $|C_{i_{min}}| = \min_{j \in [0..k] \,:\, C_j \neq \emptyset} |C_j|$, where $C_j = \{q \in T : q.color = j\}$ for every $j \in [0..k]$.*

We now consider any terminal configuration γ_t of $\mathcal{DS}(k) \circ \mathcal{ST}$ (such a configuration exists by Corollary 1 (page 37, Lemma 12 and Theorem 11). Let c_t be the unique value in the variables $\{p.min\}$ in γ_t (c_t is well-defined by Theorem 11). In γ_t, the output of $\mathcal{DS}(k) \circ \mathcal{ST}$ is the set $DS^{out} = \{p \in V : IsDominator(p)\}$.

From Theorem 11 and definition of predicate $IsDominator(p)$, we can deduce the following lemma:

Lemma 16 *In γ_t, $DS^{out} = \{r\} \cup DS^{c_t}$ where $DS^{c_t} = \{p \in V : lvl_T(p) \bmod (k+1) = c_t\}$.*

We now show that, in any case, DS^{out} is the same set as the one obtained by applying the constructive method given in the proof of Theorem 10.

We first recall some definitions. We divide the processes into sets $T_0, \ldots, T_h$ according to their level in the tree, and assign all the processes of level i to T_i. These sets are merged into $k+1$ sets $D_0, \ldots, D_k$ by taking $D_i = \bigcup_{j \geq 0} T_{i+j(k+1)}$.

Remark 3 $DS^{c_t} = D_{c_t}$.

Theorem 12 *In γ_t, DS^{out} is a k-dominating set of G, and $|DS^{out}| \leq \lceil \frac{n}{k+1} \rceil$.*

Proof. We have three cases.

- $k \geq h(T)$. In this case, the proof of Theorem 10 states that D_0 is a k-dominating set of size at most $\lceil \frac{n}{k+1} \rceil$. By Theorem 11.(b), c_t is the smallest index in $[0..k]$ such that $|C_{c_t}| = \min_{j \in [0..k] \,:\, C_j \neq \emptyset} |C_j|$, where $C_j = \{q \in T : q.color = j\}$ for every $j \in [0..k]$. Moreover, by Theorem 11.(a), $C_j = D_j$ for every $j \in [0..k]$. Thus, c_t is the smallest index in $[0..k]$ such that $|D_{c_t}| = \min_{j \in [0..k] \,:\, D_j \neq \emptyset} |D_j|$. By definition, $\min_{j \in [0..k] \,:\, D_j \neq \emptyset} |D_j| \geq 1$. Now, as $k \geq h(T)$, $D_0 = \{r\}$, *i.e.*, $|D_0| = 1$ and $c_t = 0$. Hence, $DS^{c_t} = D_0$ by Remark 3, and $DS^{out} = \{r\} \cup D_0 = D_0$, and we are done.

- $k < h(T)$ *and for every* $i \in [0..k-1]$, $|D_i| = |D_{i+1}|$. The proof is similar to that of the previous case.

- $k < h(T)$ *and there exists* $i \in [0..k-1]$ *such that* $|D_i| \neq |D_{i+1}|$. Let i_{min} the smallest index such that $|D_{i_{min}}| = \min_{j \in [0..k] \,:\, D_j \neq \emptyset} |D_j|$. In this case, the proof of Theorem 10 states that $\{r\} \cup D_{i_{min}}$ is a k-dominating set of size at most $\lceil \frac{n}{k+1} \rceil$. By Theorem 11.(b), c_t is the smallest index in $[0..k]$ that satisfies $|C_{c_t}| = \min_{j \in [0..k] \,:\, C_j \neq \emptyset} |C_j|$ where $C_j = \{q \in T : q.color = j\}$ for every $j \in [0..k]$.

  Moreover, by Theorem 11.(a), $C_j = D_j$ for every $j \in [0..k]$. Thus, c_t is the smallest index in $[0..k]$ such that $|D_{c_t}| = \min_{j \in [0..k] \,:\, D_j \neq \emptyset} |D_j|$. Hence, $c_t = i_{min}$, $DS^{c_t} = D_{i_{min}}$ by Remark 3, $DS^{out} = \{r\} \cup D_{i_{min}}$, and we are done.

In all cases, DS^{out} is equal to the set obtained by applying the constructive method given in the proof of Theorem 10. Hence, the theorem holds. □

From Theorems 11 and 12, we can deduce the following theorem:

Theorem 13 *Starting from any configuration where no action of* $\mathcal{ST}$ *is enabled, algorithm* $\mathcal{DS}(k)$ *converges, in at most* $3n$ *rounds to a terminal configuration satisfying* $Spec_{\mathcal{DS}(k)}$.

From Corollary 1 (page 37), Lemma 12 and Theorem 13, we can deduce the following theorem:

Theorem 14 $\mathcal{DS}(k) \circ \mathcal{ST}$ *is silent and stabilizes with respect to* $Spec_{\mathcal{DS}(k)}$ *within* $O(n)$ *rounds, under a weakly fair daemon.*

Figure 7.2 shows an example of a 2-dominating set computed by $\mathcal{DS}(2) \circ \mathcal{ST}$. In the figure, bold lines represent tree-edges, and dashed lines indicate non-tree-edges. In this example, $r.pop[0] = 5$, $r.pop[1] = 5$ and $r.pop[2] = 3$ once $\mathcal{DS}(2) \circ \mathcal{ST}$ stabilizes. Thus, $r.min = 2$, which means that the smallest color in use is 2. $D_2 = \{p_4, p_9, p_{10}\}$ and $|D_2| = 3$. In this case, the 2-dominating set that $\mathcal{DS}(2) \circ \mathcal{ST}$ eventually outputs is $SD = \{r\} \cup D_2$, *i.e.*, $\{r, p_4, p_9, p_{10}\}$. This 2-dominating set follows the bound given in Theorem 10, as the size of SD is 4, which is less than $\lceil \frac{13}{2+1} \rceil = 5$. However, SD is not minimal. For example, $\{r, p_{10}\}$ is a proper subset of SD that is 2-dominating, and is in fact minimal.

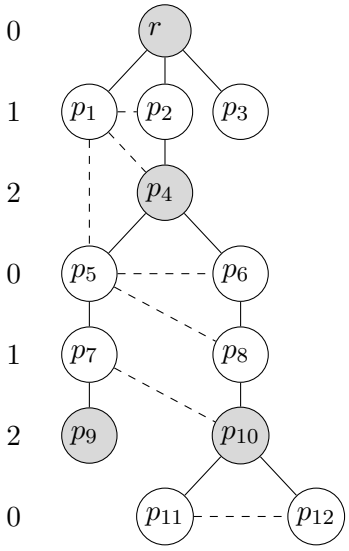


Figure 7.2 – Example of 2-dominating set computed by Algorithm $\mathcal{DS}(2) \circ \mathcal{ST}$.

7.2.3 Algorithm $\mathcal{MIN}(k)$

$\mathcal{MIN}(k)$ is also silent and computes a minimal k-dominating set which is a subset of the k-dominating set computed by $\mathcal{DS}(k)$. In Chapter 9, we will see that the minimization performed by $\mathcal{MIN}(k)$ provides an improvement which is not negligible.

This last layer of our algorithm can be achieved using the silent self-stabilizing algorithm $\mathcal{MIN}(k)$ given in [DDL09]. This algorithm takes a k-dominating set I as input, and constructs a subset of I that is a minimal k-dominating set. The knowledge of I is distributed meaning that every process p uses only the input $IsDominator(p)$ to know whether it is in the k-dominating set or not. Based on this input, $\mathcal{MIN}(k)$ assigns the output Boolean variable $p.inD$ of every process p in such way that eventually $\{p \in V : p.inD = \text{TRUE}\}$ is a minimal k-dominating set of the network.

Using the output of algorithm $\mathcal{DS}(k) \circ \mathcal{ST}$ as input for algorithm $\mathcal{MIN}(k)$, the size of the resulting minimal k-dominating set remains bounded by $\lceil \frac{n}{k+1} \rceil$, because $\mathcal{MIN}(k)$ can only remove processes in the k-dominating set computed by $\mathcal{DS}(k)$. Hence, $\mathcal{MIN}(k) \circ \mathcal{DS}(k) \circ \mathcal{ST}$ stabilizes w.r.t. the predicate $Spec_{\mathcal{SMDS}(k)}$, which is the conjunction of the following two conditions:

1. The configuration is terminal.

2. The set $\{p \in V : p.inD = \text{TRUE}\}$ is a minimal k-dominating size at most $\lceil \frac{n}{k+1} \rceil$.

As $\mathcal{SMDS}(k) = \mathcal{MIN}(k) \circ \mathcal{DS}(k) \circ \mathcal{ST}$, from Corollary 1 (page 37) and Theorem 14, we have:

Theorem 15 (Overall Correctness) *$\mathcal{SMDS}(k)$ is silent, and stabilizes with respect to $Spec_{\mathcal{SMDS}(k)}$ under a weakly fair daemon.*

7.2.4 Complexity Analysis

We first consider the round complexity of $\mathcal{SMDS}(k)$. Using the algorithm of [DLV11b], $\mathcal{ST}$ stabilizes in $O(n)$ rounds. Once the spanning tree is available, $\mathcal{DS}(k)$ stabilizes in $O(n)$ rounds, by Theorem 14. Finally, the k-dominating set computed by the first two layers is minimized by $\mathcal{MIN}(k)$ in $O(n)$ rounds, according to [DDL09]. Thus, we have:

Theorem 16 *$\mathcal{SMDS}(k)$ stabilizes with respect to $Spec_{\mathcal{SMDS}(k)}$ in $O(n)$ rounds.*

We now consider the space complexity of $\mathcal{SMDS}(k)$. $\mathcal{ST}$ and $\mathcal{MIN}(k)$ can be implemented using $O(\log n)$ bits per process [DLV11b, DDL09]. $\mathcal{DS}(k)$ at each process is composed of two variables whose domain has $k+1$ elements, and an array of $k+1$ integers. However, in the terminal configuration, the minimum non-null value of a cell is at most $\lceil \frac{n}{k+1} \rceil$. Thus, the algorithm still works if we replace any assignment of any value val to a cell by $\min(val, \lceil \frac{N}{k+1} \rceil + 1)$, where N is any upper bound on n. In this case, each array can be implemented using $O(k \log \frac{N}{k})$ bits. Note that this bound can be obtained only if we assume that each process knows the upper bound N. However, n can be computed dynamically using the spanning tree.

Theorem 17 *$\mathcal{SMDS}(k)$ can be implemented using $O(\log k + \log n + k \log \frac{N}{k})$ bits per process, where N is any upper bound on n.*

Previously in this section, we show that $\mathcal{SMDS}(k)$ stabilizes w.r.t. $Spec_{\mathcal{SMDS}(k)}$ under a weakly fair daemon. We now use the automatic method given in Section 3.6 (page 37) to transform algorithm $\mathcal{SMDS}(k)$, which stabilizes with respect to $Spec_{\mathcal{SMDS}(k)}$ under a weakly fair daemon as proven hitherto, into a self-stabilizing algorithm $\mathcal{SMDS}(k)^t$ which works under an unfair daemon (for the same specification). Note that the silence property of $\mathcal{SMDS}(k)$ is still preserved for $\mathcal{SMDS}(k)^t$ the input algorithm. Then, $\mathcal{SMDS}(k)^t$ stabilizes w.r.t. $Spec_{\mathcal{SMDS}(k)}$ in $O(n)$ rounds and $O(\mathcal{D}n^3)$ steps using $O(\log k + \log n + k \log \frac{N}{k})$ bits per process, by Theorems 16-17 and Theorems 4-6 (pages 39-40). This illustrates how our transformer does not degrade the round complexity and memory requirements while achieving an interesting step complexity.

Algorithm 4 $\mathcal{DS}(k)$, code for each process p

Inputs:

$\texttt{Parent}(p) \in \mathcal{N}(p) \cup \{\perp\}$	Parent process of p in the spanning tree, $\perp$ for the root.

Variables:

$p.color \in [0..k]$	Color of p.
$p.pop[i]$ for all $i \in [0..k]$	Population, *i.e.*, number of processes, of color i in $T(p)$.
$p.min \in [0..k]$	Color with the smallest population in $T(p)$.

Macros:

$$
\begin{array}{lcl}
\texttt{EvalColor}(p) & = & 0 \textbf{ if } (\texttt{Parent}(p) = \perp) \textbf{ else } (\texttt{Parent}(p).color + 1) \bmod (k+1) \\
\texttt{SelfPop}(p,i) & = & 1 \textbf{ if } (p.color = i) \textbf{ else } 0 \\
\texttt{Children}(p) & = & \{q \in \mathcal{N}(p) : \texttt{Parent}(q) = p\} \\
\texttt{EvalPop}(p,i) & = & \texttt{SelfPop}(p,i) + \sum_{q \in \texttt{Children}(p)} q.pop[i] \\
\texttt{MinPop}(p) & = & \min_{i \in [0..k]} \{p.pop[i] : p.pop[i] > 0\} \\
\texttt{MinColor}(p) & = & \min_{i \in [0..k]} \{i : p.pop[i] = \texttt{MinPop}(p)\} \\
\texttt{EvalMin}(p) & = & \texttt{MinColor}(p) \textbf{ if } (\texttt{Parent}(p) = \perp) \textbf{ else } \texttt{Parent}(p).min
\end{array}
$$

Predicates:

$$
\begin{array}{lcl}
IsRoot(p) & \equiv & \texttt{Parent}(p) = \perp \\
ColorOK(p) & \equiv & p.color = \texttt{EvalColor}(p) \\
PopOK(p) & \equiv & \forall i \in [0..k], p.pop[i] = \texttt{EvalPop}(p,i) \\
MinOK(p) & \equiv & p.min = \texttt{EvalMin}(p) \\
IsDominator(p) & \equiv & IsRoot(p) \vee (p.color = p.min)
\end{array}
$$

Actions:

$$
\begin{array}{llcll}
\text{(1) FixColor} & :: & \neg ColorOK(p) & \longrightarrow & p.color \leftarrow \texttt{EvalColor}(p) \\
\text{(2) FixPop} & :: & \neg PopOK(p) & \longrightarrow & \forall i \in [0..k], p.pop[i] \leftarrow \texttt{EvalPop}(p,i) \\
\text{(3) FixMin} & :: & \neg MinOK(p) & \longrightarrow & p.min \leftarrow \texttt{EvalMin}(p)
\end{array}
$$

Chapter 8

Competitive k-Clustering

Contents

In this chapter, we give a silent self-stabilizing algorithm for constructing a k-clustering of any connected network under a *weakly fair* daemon. Our algorithm stabilizes in $O(n)$ rounds using $O(\log k + \log n)$ space per process.

In the general case, our algorithm constructs at most $\lceil \frac{n}{k+1} \rceil$ k-clusters. In the case where the network is a unit-disk graph (UDG), then our algorithm is $7.2552k + O(1)$-*competitive*, that is, it builds a k-clustering which has at most $7.2552k + O(1)$ times as many clusters as the minimum cardinality k-clustering of this network. More generally, if the network is an approximate disk graph (ADG) with approximation ratio λ, then our algorithm is $7.2552\lambda^2 k + O(\lambda)$-competitive. Finally, in the case of a tree network, our algorithm computes a k-clustering with the minimum number of clusters.

Roadmap. In the next section, we present our silent self-stabilizing algorithm, called $\mathcal{CLR}(k)$, which constructs a k-clustering in any directed tree T. By composing $\mathcal{CLR}(k)$ with any silent self-stabilizing spanning tree algorithm, we obtain a silent self-stabilizing k-clustering algorithm that builds a k-clustering in any arbitrary network.

Then, we prove the correctness of our algorithm in Section 8.2. We also establish that it stabilizes in $O(h)$ rounds, where h is the height of T, under a weakly fair daemon. At the end of that section, we show that the composition of $\mathcal{CLR}(k)$ with any silent self-stabilizing spanning tree algorithm computes a k-clustering with at most $\lceil \frac{n}{k+1} \rceil$ distinct k-clusters in arbitrary networks.

In Section 8.3, we show that the k-clustering computed by $\mathcal{CLR}(k)$ is optimal in any tree network.

Finally, we will see in Section 8.4 that the composition between $\mathcal{CLR}(k)$ and our spanning tree construction $\mathcal{MIST} \circ \mathcal{BFST}$ given in Chapter 4 is competitive in both UDG and ADG networks. The stabilization time of $\mathcal{CLR}(k) \circ \mathcal{MIST} \circ \mathcal{BFST}$ is $O(n)$ rounds and its memory requirement is $O(\log k + \log n)$ space per process.

Note that in the latter case, the MIS tree construction algorithm is the time bottleneck of our k-clustering algorithm, as it takes $O(n)$ rounds, and the remainder of the algorithm takes $O(\mathcal{D})$ rounds, where $\mathcal{D}$ is the diameter of the network. We would like to improve that time to be $O(\mathcal{D})$, however, that will most likely involve different techniques, since whether a given process is part of the Fernandess-Malkhi MIS is a $\mathcal{P}$-*complete* problem, as established by Theorem 8 at page 54.

8.1 Algorithm $\mathcal{CLR}(k)$

In this section, we give the formal description of $\mathcal{CLR}(k)$ in Algorithm 5 and below the intuitive ideas behind it.

$\mathcal{CLR}(k)$ builds a k-clustering in two phases. During the first phase, $\mathcal{CLR}(k)$ computes the set of clusterheads, Dom, which has cardinality at most $\lceil \frac{n}{k+1} \rceil$. The second phase consists of building a spanning forest, where each directed tree is rooted at a clusterhead and represents the k-cluster of that clusterhead. Hence, we obtain a k-clustering of at most $\lceil \frac{n}{k+1} \rceil$ k-clusters. $\mathcal{CLR}(k)$ uses the following three variables in the code of each process p:

- $p.\alpha$, *an integer in the range* $[0..2k]$. In any terminal configuration, the set of clusterheads Dom is defined as the set of processes p such that $p.\alpha = k$ or $p.\alpha < k$ and $p = r$, that is, such that `IsClusterHead`(p) = TRUE.
- $p.\text{parent}_{CLR} \in \mathcal{N}(p) \cup \{p\}$. In any terminal configuration, $p.parent_{CLR}$ is the parent of p in its k-cluster, unless p is a clusterhead, in which case $p.parent_{CLR} = p$.
- $p.\text{head}_{CLR} \in V$. In any terminal configuration, $p.head_{CLR}$ is equal to the identifier of the clusterhead in the k-cluster that p belongs to.

Building Dom**.** The first phase of $\mathcal{CLR}(k)$ consists of building the set Dom as a k-dominating set of T, that is, a subset of processes such that every process is at most at distance k from a process in Dom. Dom is constructed by dynamic programming, starting from the leaves of T. As previously explained, Dom is defined using the values of $p.\alpha$ for all p.

Consider any terminal configuration. In this configuration, $p.\alpha = \|p, q\|$, where q is the furthest process in $T(p)$ that is in the same k-cluster as p. See Figure 8.1.

(i) If $p.\alpha < k$, that is, p satisfies `IsShort`(p), then p is said to be *short* and we have two cases: $p \neq r$ or $p = r$. In the former case, p is k-dominated by a process of Dom outside of its subtree, that is, the path from p to its clusterhead goes through the parent link of p in the tree, and the distance to this process is at most $k - p.\alpha$ (see Figure 8.3). In the latter case, p is not k-dominated by any other process of Dom inside its subtree and, by definition, there is no process outside its subtree (see the root in Figure 8.3.(b)). Thus, p must be placed in Dom.

Algorithm 5 $\mathcal{CLR}(k)$, code for each process p

Input:

$\texttt{Parent}(p) \in \mathcal{N}(p) \cup \{p\}$

Variables:

$p.\alpha \in [0..2k]$
$p.parent_{CLR} \in \mathcal{N}(p) \cup \{p\}$
$p.head_{CLR} \in V$

Macros:

$\texttt{IsShort}(p) \equiv p.\alpha < k$
$\texttt{IsTall}(p) \equiv p.\alpha \geq k$
$\texttt{IsClusterHead}(p) \equiv (p.\alpha = k) \vee (\texttt{IsShort}(p) \wedge (p = r))$
$\texttt{ShortChildren}(p) = \{q \in \mathcal{N}(p) : (\texttt{Parent}(q) = p) \wedge \texttt{IsShort}(q)\}$
$\texttt{TallChildren}(p) = \{q \in \mathcal{N}(p) : (\texttt{Parent}(q) = p) \wedge \texttt{IsTall}(q)\}$
$\texttt{MaxAShort}(p) =$ **if** $\texttt{ShortChildren}(p) = \emptyset$ **then** -1 **else** $\max\{q.\alpha : q \in \texttt{ShortChildren}(p)\}$
$\texttt{MinATall}(p) =$ **if** $\texttt{TallChildren}(p) = \emptyset$ **then** $2k+1$ **else** $\min\{q.\alpha : q \in \texttt{TallChildren}(p)\}$
$\texttt{MinIDMinATall}(p) =$ **if** $\texttt{TallChildren}(p) = \emptyset$ **then** p **else** $\min\{q \in \texttt{TallChildren}(p) : q.\alpha = \texttt{MinATall}(p)\}$
$\texttt{Alpha}(p) =$ **if** $\texttt{MaxAShort}(p) + \texttt{MinATall}(p) \leq 2k-2$ **then** $\texttt{MinATall}(p)+1$ **else** $\texttt{MaxAShort}(p)+1$
$\texttt{Parent}_{CLR}(p) =$ **if** $\texttt{IsClusterHead}(p)$ **then** p **else if** $p.\alpha < k$ **then** $\texttt{Parent}(p)$ **else** $\texttt{MinIDMinATall}(p)$
$\texttt{Head}_{CLR}(p) =$ **if** $\texttt{IsClusterHead}(p)$ **then** p **else** $p.parent_{CLR}.head_{CLR}$

Actions:

(1) $\texttt{SetAlpha} :: p.\alpha \neq \texttt{Alpha}(p) \longrightarrow p.\alpha \leftarrow \texttt{Alpha}(p)$
(2) $\texttt{SetParent} :: p.parent_{CLR} \neq \texttt{Parent}_{CLR}(p) \longrightarrow p.parent_{CLR} \leftarrow \texttt{Parent}_{CLR}(p)$
(3) $\texttt{SetHead} :: p.head_{CLR} \neq \texttt{Head}_{CLR}(p) \longrightarrow p.head_{CLR} \leftarrow \texttt{Head}_{CLR}(p)$

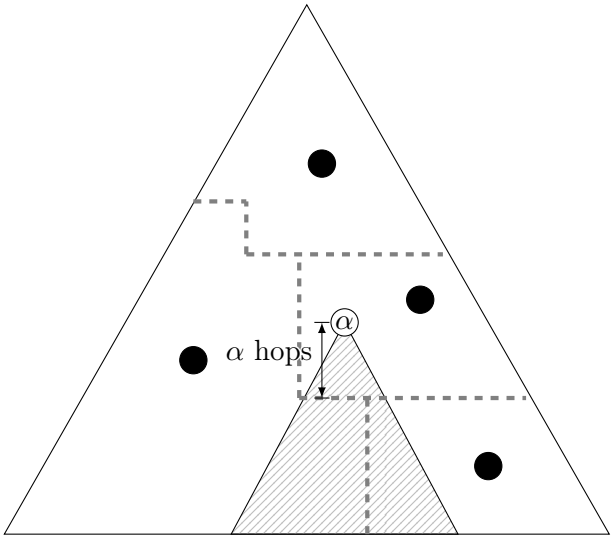


Figure 8.1 – Value of $p.\alpha$: Dashed lines represent cluster boundaries, black processes are clusterheads, and the shade area represents the subtree of the white process.

(*ii*) If $p.\alpha \geq k$, that is, p satisfies `IsTall`(p), then p is said to be *tall* and there is a process q at $p.\alpha - k$ hops below p such that $q.\alpha = k$ (see Figure 8.3). So, $q \in Dom$ and p is k-dominated by q. Note that, if $p.\alpha = k$, then $p.\alpha - k = 0$, that is, $p = q$ and p belongs to Dom.

$p.\alpha$ is computed using `Alpha`(p) which is based on the two following macros:

- `MaxAShort`(p) returns the maximum value of $q.\alpha$ for all *short* children q of p. If p has no *short* child, `MaxAShort`(p) returns -1.
- `MinATall`(p) returns the minimum value of $q.\alpha$ for all *tall* children q of p. If p has no *tall* child, `MinATall`(p) returns $2k + 1$.

According to these macros, $p.\alpha$ is computed by Action `SetAlpha` in a bottom-up fashion in the tree T as follows:

- If `MaxAShort`(p) + `MinATall`(p) $> 2k - 2$, $p.\alpha =$ `MaxAShort`(p) $+ 1$.
- If `MaxAShort`(p) + `MinATall`(p) $\leq 2k - 2$, $p.\alpha =$ `MinATall`(p) $+ 1$.

Consider a leaf f. By definition, `MaxAShort`(f) + `MinATall`(f) $= -1 + 2k + 1 > 2k - 2$. Thus, $f.\alpha = -1 + 1 = 0$, which corresponds to the distance between f and its furthest descendant that will be in its cluster (f itself).

Consider now an internal process p and assume that the α-variables of all its children are correctly evaluated. p should choose a clusterhead that will be either (1) in its subtree (in this case, p will be *tall*), or (2) outside its subtree (in this case p will be *short*). We should preferably make the choice (1) to reduce the number of clusterheads.

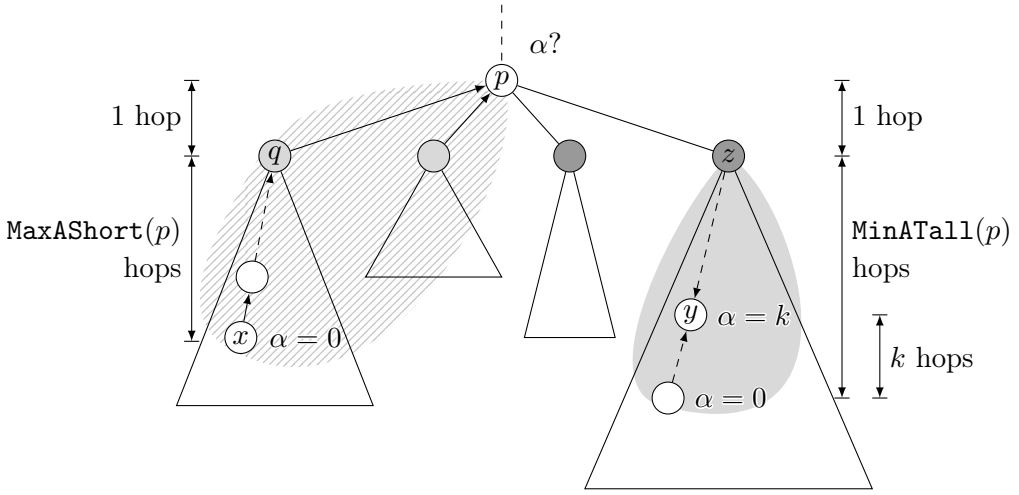


Figure 8.2 – Illustrative example: Light-gray processes are *short* children of p; gray processes are *tall* children of p. The shade area shows the processes that already choose the same cluster as p. The light-gray area shows the processes that already choose the same cluster as z.

Let q be a *short* child of p. From (i), the path from q to its clusterhead goes through p. Thus, to prevent cycle creation, $(*)$ p should not choose a clusterhead that is in the subtree of any of its short children.

From now on, follow the illustrative example given in Figure 8.2. Let x be the furthest process that is both in the subtree of some *short* child of p and in the same cluster as p. Let q be the *short* child of p such that $x \in T(q)$. Then, from (i), x is at distance $\texttt{MaxAShort}(p) + 1$ from p. Two cases are then possible:

- $\texttt{MaxAShort}(p) + \texttt{MinATall}(p) > 2k - 2$. If p chooses a process y of its subtree as clusterhead, then from $(*)$ above, the path from p to its clusterhead should go through one of its *tall* child. So, p will be at least at distance $\texttt{MinATall}(p) - k + 1$ from that clusterhead, from (ii). Now, in this case, x will be at least at distance $\texttt{MaxAShort}(p) + 1 + \texttt{MinATall}(p) - k + 1 > 2k - 2 - k + 2 = k$ from the clusterhead y, this violates the definition of k-clustering. Thus, p should necessarily choose its clusterhead outside the subtrees of any of its children (that is, either p declares itself as clusterhead or chooses an ancestor as clusterhead). From (i) and (ii), this in particular means that all processes in the subtrees of the *tall* children of p adopt a different cluster from p and, consequently the process x is then the furthest process that belongs to both $T(p)$ and the cluster of p. This implies that $p.\alpha = \|p, x\| = \texttt{MaxAShort}(p) + 1$.

- $\texttt{MaxAShort}(p) + \texttt{MinATall}(p) \leq 2k - 2$. Let z be a *tall* child of p such that $z.\alpha = \texttt{MinATall}(p)$. Unlike the previous case, p can choose a process y in the subtree of z as clusterhead. Indeed, in this case, x will be at distance $\texttt{MaxAShort}(p) + 1 + \texttt{MinATall}(p) - k + 1 \leq 2k - 2 - k + 2 = k$ from y. Hence, the processes (other than p) that are both in the subtree of p and in its cluster

will be either processes in subtrees of short children of p or processes in $T(z)$. Since by definition, $\texttt{MinATall}(p) > \texttt{MaxAShort}(p)$, the furthest process that belongs to both $T(p)$ and the cluster of p will be at distance $\texttt{MinATall}(p) + 1$ from p, *i.e.*, $p.\alpha = \texttt{MinATall}(p) + 1$.

To help the reader's intuition, we summarize below the important properties of $p.\alpha$, for any process p. These properties can be checked in the examples given in Figure 8.3, and will be proven in Section 8.2.

Property 3 *In any terminal configuration, for every process p, we have:*

(a) If $p.\alpha > 0$, then there is some child q of p such that $q.\alpha = p.\alpha - 1$.

(b) If $p.\alpha > k$, then there is a proper descendant q of p such that $q \in Dom$ and q is $p.\alpha - k$ levels below p.

(c) There is a member of Dom within $|p.\alpha - k|$ hops of p.

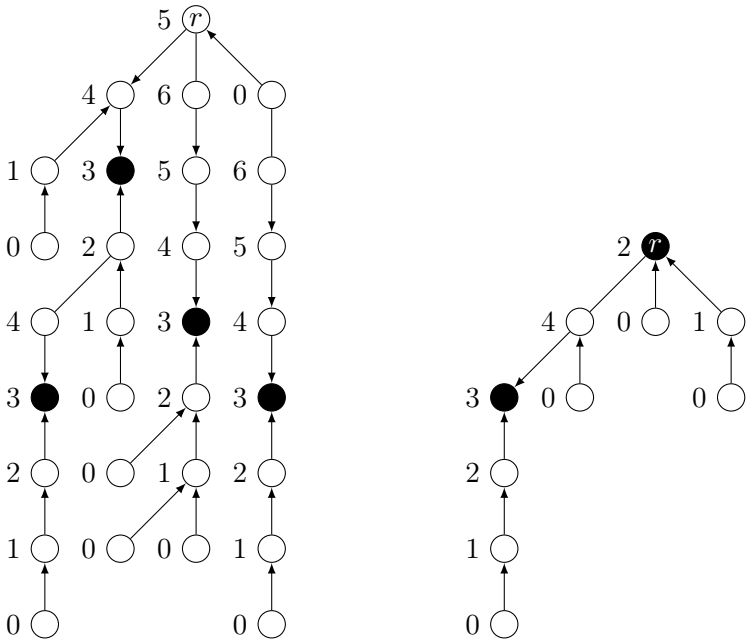
Constructing the k-Clustering. The second phase of $\mathcal{CLR}(k)$ partitions the processes into distinct k-clusters, each of which contains one clusterhead. Each k-cluster contains a *k-cluster spanning tree*, a tree containing all the processes of that k-cluster. Each k-cluster spanning tree is a subgraph of T rooted at the clusterhead, possibly with the directions of some edges reversed. Furthermore, the height of the k-cluster spanning tree is at most k.

Each process of Dom designates itself as clusterhead using Actions `SetParent` and `SetHead`. Other processes p designate their parent (using Action `SetParent`) as follows: (1) if p is *short*, then its parent in its k-cluster is its parent in the tree; (2) if p is *tall*, then p selects as parent in its k-clustering its *tall* child in the tree of minimum α value (we use UIDs to break ties, see $\texttt{MinIDMinATall}(p)$). Finally, identifiers of clusterheads are propagated in a top-down fashion in their k-cluster using Action `SetHead`.

Two examples of 3-clustering using $\mathcal{CLR}(3)$ are given in Figure 8.3. In Figure 8.3a, the root is a *tall* process, thus it is not a clusterhead. On the contrary, in Figure 8.3b, the root is a *short* process, consequently it is a clusterhead.

8.2 Correctness of $\mathcal{CLR}(k)$

We first show the convergence of $\mathcal{CLR}(k)$ from any configuration to a terminal one. Since computation of the $p.\alpha$ is bottom-up in T, the time required for those values to stabilize is $O(h)$ rounds, where h is the height of T. After that, one additional round is necessary to fix the $\texttt{Parent}_{CLR}$ variables, because the values of these variables only depend on the α variables. Finally, the $head_{CLR}$ variables are fixed top-down within the k-cluster spanning trees starting from the clusterheads in $O(h)$ rounds. Hence, it follows that the time complexity of $\mathcal{CLR}(k)$ is $O(h)$ rounds, as shown below.



(a) 3-clustering where r is *tall* (b) 3-clustering where r is *short*

Figure 8.3 – Examples of k-clustering using $\mathcal{CLR}(k)$, where $k = 3$. Values of α are indicated on the left of each process, clusterheads are colored in black, and arrows represent local spanning tree of each k-cluster.

Lemma 17 *For every process p, the variable $p.\alpha$ is fixed forever within $h+1$ rounds.*

Proof. We prove this lemma by backwards induction on the level $lvl(p)$ of processes p in the tree.

As a base case, if $lvl(p) = h$, that is p is a leaf, then $p.\alpha$ is fixed forever within one round.

Assume for every p such that $lvl(p) = \ell$, the variable $p.\alpha$ is fixed forever within $h - \ell + 1$ rounds.

Let q be a process such that $lvl(q) = \ell - 1$. The value of $\texttt{Alpha}(q)$ depends only on the values of every $p.\alpha$ where p has level ℓ. By the induction hypothesis, all those values are fixed within $h - \ell + 1$ rounds, thus $q.\alpha$ is fixed within one additional round, that is within $h - \ell + 2 = h - (\ell - 1) + 1$ rounds.

This complexity is maximum with $\ell = 0$ and the lemma follows. □

Lemma 18 *For every process p, the variable $p.\text{parent}_{CLR}$ is fixed forever within $h + 2$ rounds.*

Proof. The evaluation of both guard and statement of Action `SetParent` only relies, for a process p, on the variables $p.parent_{CLR}$ and $q.\alpha$ for every neighbor q of p. Thus, after all α variables are fixed in the network, every $p.parent_{CLR}$ is fixed within one additional round. By Lemma 17, we are done. □

Lemma 19 *In every configuration where all* parent_{CLR} *and* α *variables are fixed forever, there is no directed cycle constituted of directed edges of the form* $(p, p.\text{parent}_{CLR})$ *except self-loops.*

Proof. The network being a tree, we only need to exclude the existence of cycle of size two. Assume by the contradiction that such a cycle exists between p and its neighbor q, that is $p.parent_{CLR} = q$ and $q.parent_{CLR} = p$. Without loss of generality, assume that q is a child of p. Then, by definition of Macro $\texttt{Parent}_{CLR}(q)$, $q.\alpha < k$. By definition of Macro $\texttt{Parent}_{CLR}(p)$, $q.\alpha \geq k$, a contradiction. □

Lemma 20 *For every process* p*, the variable* $p.\text{head}_{CLR}$ *is fixed forever within* $O(h)$ *rounds.*

Proof. By Lemmas 17 and 18, the variables $p.\alpha$ and $p.parent_{CLR}$ are fixed within $h + 2$ rounds.

Then, for every process p, the variable $p.head_{CLR}$ only depends on $p.parent_{CLR}.head_{CLR}$ and some fixed variables.

For every process p such that $p.parent_{CLR} = p$, $p.head_{CLR}$ is fixed forever in at most one additional round. Then, changes on $head_{CLR}$ can be propagated from process p to its neighbor q only if $q.parent_{CLR} = p$. By Lemma 19, these propagations end after $O(h)$ rounds, and we are done. □

From Lemmas 17 to 20, follows:

Lemma 21 *Starting from any configuration,* $\mathcal{CLR}(k)$ *reaches a terminal configuration in* $O(h)$ *rounds.*

We now consider any terminal configuration of $\mathcal{CLR}(k)$ and show that such a configuration is legitimate. The proof begins by formally establishing the three claims given in Property 3, respectively using Remark 4, Lemma 22, and Lemma 23.

Remark 4 *Property 3.(a) follows immediately from the definition of* α*.*

Below, we prove Property 3.(b).

Lemma 22 *In any terminal configuration of* $\mathcal{CLR}(k)$*, for every process* p*, if* $p.\alpha > k$*, then there is a proper descendant* q *of* p *such that* $q \in Dom$ *and* q *is* $p.\alpha - k$ *levels below* p*.*

Proof. We prove this lemma by strong induction on $p.\alpha$.

As a base case, if $p.\alpha = k + 1$, then, by Property 3.(a), there is a child q of p such that $q.\alpha = k$, that is $q \in Dom$.

Assume the lemma holds for every p such that $k < p.\alpha < a$ and let p' be a process such that $p'.\alpha = a$.

By Property 3.(a), there is a child q' of p' such that $q'.\alpha = p'.\alpha - 1$. By the induction hypothesis, there is a proper descendant q'' of q' such that $q'' \in Dom$ and

q'' is $q'.\alpha - k$ levels below q'. So, q'' is $q'.\alpha - k + 1 = p'.\alpha - 1 - k + 1 = p'.\alpha - k$ below p', and we are done. □

We now prove Property 3.(c).

Lemma 23 *In any terminal configuration of $\mathcal{CLR}(k)$, for every process p, there is a process q such that $q \in Dom$ and $\|p, q\| \leq |p.\alpha - k|$.*

Proof. If $p.\alpha > k$, then, by Lemma 22, we are done.

Consider now any process p such that $p.\alpha \leq k$. We prove the lemma by strong backward induction on $p.\alpha$.

As a base case, if $p.\alpha = k$, then $p \in Dom$ by definition.

Assume the lemma holds for every p' such that $a < p'.\alpha \leq k$.

Let q be a process such that $q.\alpha = a$ and $q \neq r$. Indeed, if $r.\alpha \leq k$, then $r \in Dom$ by definition. Let q' be the parent of q. We consider two cases.

- Assume $q'.\alpha = \texttt{MaxAShort}(q') + 1$. As $q.\alpha < k$, q is *short* and $q.\alpha \leq \texttt{MaxAShort}(q')$. So:

$$\begin{array}{ll} q.\alpha & < q'.\alpha \leq k \\ a & < q'.\alpha \leq k \end{array}$$

  By the induction hypothesis, there is a member of Dom which is within $k - q'.\alpha$ hops of q'. This process is within $k - q'.\alpha + 1$ hops from q. Now:

$$\begin{array}{ll} a & < q'.\alpha \\ -q'.\alpha & < -a \\ k - q'.\alpha + 1 & < k - a + 1 \\ k - q'.\alpha + 1 & \leq k - a \\ k - q'.\alpha + 1 & \leq k - q.\alpha \\ k - q'.\alpha + 1 & \leq |q.\alpha - k| \end{array}$$

  This process is within $|q.\alpha - k|$ hops from q and we are done.

- Otherwise, $q'.\alpha = \texttt{MinATall}(q') + 1$ and $q'.\alpha > k$. By Lemma 22, there is some $q'' \in Dom$ within $q'.\alpha - k$ hops of q'. Thus, $\|q'', q\| \leq q'.\alpha - k + 1$. Then, by definition of α:

$$\begin{array}{ll} \texttt{MaxAShort}(q') + \texttt{MinATall}(q') & \leq 2k - 2 \\ \texttt{MinATall}(q') - k + 2 & \leq k - \texttt{MaxAShort}(q') \\ q'.\alpha - k + 1 & \leq k - q.\alpha \end{array}$$

  Hence:

$$\begin{array}{ll} \|q'', q\| & \leq k - q.\alpha \\ \|q'', q\| & \leq |q.\alpha - k| \end{array}$$

  So, q'' is within $|q.\alpha - k|$ hops from q and we are done.

□

We now use Property 3 to complete the correctness proof of $\mathcal{CLR}(k)$.

Since $|p.\alpha - k| \leq k$ for every p, we can deduce the following corollary from Property 3.(c).

Corollary 2 *In any terminal configuration of $\mathcal{CLR}(k)$, Dom is a k-dominating set of T.*

The following lemma shows that every process is in the k-cluster of a member of *Dom*.

Lemma 24 *In any terminal configuration of $\mathcal{CLR}(k)$, for every process p, there is a path $P = (p_0 = p, \ldots, p_m)$ such that: (1) $m \leq |p.\alpha - k| \leq k$, (2) $\forall i \in [0..m-1], p_i.\text{parent}_{CLR} = p_{i+1}$, (3) $p_m.\text{parent}_{CLR} = p_m$, (4) $\forall i \in [0..m], p_i.\text{head}_{CLR} = p_m$, (5) $p_m \in Dom$.*

Proof. We prove this lemma by strong induction on $|p.\alpha - k|$. Note that $p.\alpha \in [0..2k]$, thus $|p.\alpha - k| \in [0..k]$ always.

As a base case, if $p.\alpha = k$, then $\texttt{IsClusterHead}(p) = \text{TRUE}$. Thus, by definition, $p.parent_{CLR} = p$ and $p.head_{CLR} = p$. The path $P = (p)$ verifies each property stated in the lemma.

Assume the lemma holds for every q such that $|q.\alpha - k| < a$, and consider a process p such that $|p.\alpha - k| = a$.

If $p.\alpha > k$, then, by definition of $Alpha(p)$, $p.\alpha = \texttt{MinATall}(p) + 1$, *i.e.*, there is some neighbor q of p such that $q.\alpha = \texttt{MinATall}(p)$, hence $p.\alpha = q.\alpha + 1$. Consider the process of smallest identifier. Since $p.\alpha - k = a$, it follows that $q.\alpha + 1 - k = a$, that is, $q.\alpha - k = a - 1 < a$. By the induction hypothesis, there is a path $Q = (p_0 = q, \ldots, p_m)$ leading to a clusterhead p_m such that:

- $m \leq |q.\alpha - k| \leq k$,
- $\forall i \in [0..m-1], p_i.parent_{CLR} = p_i + 1$,
- $p_m.parent_{CLR} = p_m$,
- $\forall i \in [0..m], p_i.head_{CLR} = p_m$.

By definition of $\texttt{Parent}_{CLR}(p)$ and $Head_{CLR}(p)$, $p.parent_{CLR} = q$ and $p.head_{CLR} = p_m$. Then, as $q.\alpha \geq k$, $|q.\alpha - k| + 1 = |q.\alpha - k + 1| = |p.\alpha - k|$. Hence, the path $p, p_0 = q, \ldots, p_m$ has length at most $|p.\alpha - k|$, and we are done.

Otherwise, $p.\alpha < k$. If $p = r$, then $\texttt{IsClusterHead}(p) = \text{TRUE}$ and the lemma holds. Consider now the case $p \neq r$ and note $q = \texttt{Parent}(p)$. By definition of $\texttt{Parent}_{CLR}(p)$, $p.parent_{CLR} = q$. By definition of $Head_{CLR}(p)$, $p.head_{CLR} = q.head_{CLR}$. We now show that $|q.\alpha - k| < a$, *i.e.*, $|q.\alpha - k| < |p.\alpha - k|$ in order to make use of the induction hypothesis as in the previous case, thus completing the proof. Two cases have to be distinguished:

- $q.\alpha \leq k$, then, by definition of $Alpha(q)$, $q.\alpha = \texttt{MaxAShort}(q) + 1$. As p is a *short* child of q, $q.\alpha \geq p.\alpha + 1$, and $q.\alpha - k > p.\alpha - k$. Since $p.\alpha < q.\alpha \leq k$, $|q.\alpha - k| < |p.\alpha - k|$.

- $q.\alpha > k$, then, by definition of $Alpha(q)$, $q.\alpha = \texttt{MinATall}(q) + 1$ and:

$$\begin{array}{ll} \texttt{MaxAShort}(q) + \texttt{MinATall}(q) & \leq 2k - 2 \\ (\texttt{MaxAShort}(q) + 1) + (q.\alpha - k) & \leq k \end{array}$$

  Since $p.\alpha \leq \texttt{MaxAShort}(q)$, then:

$$\begin{array}{ll} (p.\alpha + 1) + (q.\alpha - k) & \leq k \\ q.\alpha - k & \leq k - p.\alpha - 1 \\ |q.\alpha - k| & < |k - p.\alpha| \\ |q.\alpha - k| & < |p.\alpha - k| \end{array}$$

□

Lemma 25 *In any terminal configuration of $\mathcal{CLR}(k)$, every k-cluster whose clusterhead is not the root contains at least a path of $k+1$ processes.*

Proof. Consider any k-cluster whose clusterhead p is not the root. Then, $p.\alpha = k$, $p.parent_{CLR} = p$, and $p.head_{CLR} = p$ by definition of $\texttt{IsClusterHead}(p)$, $\texttt{Parent}_{CLR}(p)$, and $\texttt{Head}_{CLR}(p)$. Moreover, by Property 3.(a), there is a path $(p_0, \ldots, p_k)$ such that $p_k = p$ and for every $i \in [0..k-1]$, $p_i.\alpha = p_{i+1}.\alpha - 1 = i$. By Definition of Macro $\texttt{Parent}_{CLR}(p_j)$, for every $j \in [0..k-1]$, $p_j.parent_{CLR} = p_{j+1}$. By Definition of Macro $\texttt{Head}_{CLR}(p_j)$, for every $j \in [0..k-1]$, $p_j.head_{CLR} = p_{j+1}.head_{CLR} = p_k = p$. □

Lemma 26 *In any terminal configuration of $\mathcal{CLR}(k)$, there are at most $\lceil \frac{n}{k+1} \rceil$ distinct k-clusters.*

Proof. By Lemma 25, except for the k-cluster which contains the root , every k-cluster contains at least $k+1$ processes. Thus, there are at most $1 + \left\lfloor \frac{n-1}{k+1} \right\rfloor$ k-clusters. Finally, we note that $\left(1 + \left\lfloor \frac{n-1}{k+1} \right\rfloor\right) = \left\lfloor \frac{n+k}{k+1} \right\rfloor = \left\lceil \frac{n}{k+1} \right\rceil$. □

By Corollary 2 and Lemmas 24 and 26, we have:

Lemma 27 *In any terminal configuration of $\mathcal{CLR}(k)$, T is partitioned into at most $\lceil \frac{n}{k+1} \rceil$ distinct k-clusters.*

From Lemmas 21 and 27, we have:

Theorem 18 *In any tree of n processes and height h, $\mathcal{CLR}(k)$ is a silent self-stabilizing algorithm that partitions the tree within $O(h)$ rounds into at most $\lceil \frac{n}{k+1} \rceil$ distinct k-clusters under a weakly fair daemon.*

By Corollary 1 (page 37), Theorem 7 (page 49), and Theorem 18, $\mathcal{CLR}(k) \circ \mathcal{MIST} \circ \mathcal{BFST}$ is self-stabilizing, $\mathcal{MIST} \circ \mathcal{BFST}$ stabilizes within $O(n)$ rounds, and $O(h)$ rounds later $\mathcal{CLR}(k) \circ \mathcal{MIST} \circ \mathcal{BFST}$ reaches a terminal configuration, where h is the height of T_{MIS}. Now, by Property 2 (page 50), h is bounded by $2\mathcal{D}$, where $\mathcal{D}$ is the diameter of the network. Hence, from any initial configuration, $\mathcal{CLR}(k) \circ \mathcal{MIST} \circ \mathcal{BFST}$ stabilizes in $O(n)$ rounds.

Theorem 19 *In any connected network, $\mathcal{CLR}(k) \circ \mathcal{MIST} \circ \mathcal{BFST}$ is a silent self-stabilizing algorithm that builds at most $\lceil \frac{n}{k+1} \rceil$ distinct k-clusters within $O(n)$ rounds using $O(\log k + \log n)$ space per process under a weakly fair daemon.*

8.3 Optimality of the k-Clustering in Trees

In this section, we show that the set Dom of clusterheads computed by $\mathcal{CLR}(k)$ has the *minimum* cardinality, for any tree T.

Lemma 28 *Let p any process satisfying $p.\alpha < k$ in a terminal configuration γ of $\mathcal{CLR}(k)$, every child q of p satisfies $q.\alpha \neq k$.*

Proof. Assume the contrary. Then, $\texttt{MinATall}(p) = k$. So:

$$\begin{array}{ll} \texttt{MaxAShort}(p) + \texttt{MinATall}(p) & > 2k-2 \\ \texttt{MaxAShort}(p) + k & > 2k-2 \\ \texttt{MaxAShort}(p) + k & \geq 2k-1 \\ \texttt{MaxAShort}(p) + 1 & \geq k \\ p.\alpha & \geq k \end{array}$$

Hence, we obtain a contradiction and consequently $q.\alpha \neq k$. □

Lemma 29 *In any terminal configuration γ of $\mathcal{CLR}(k)$, for every process p, for every process q in $(T(p) \cap Dom) \setminus \{p\}$, we have:*

- *If $p.\alpha \leq k$, then $\|p, q\| > |p.\alpha - k|$.*
- *If $p.\alpha > k$, then $\|p, q\| \geq |p.\alpha - k|$.*

Proof. We prove this lemma by backwards induction on the level $lvl(p)$ of processes p in the tree.

If $lvl(p) = h$, then p is a leaf and $(T(p) \cap Dom) \setminus \{p\} = \emptyset$, so the lemma trivially holds.

Assume the lemma holds for every process x such that $\ell < lvl(x) \leq h$ and let p be a process such that $lvl(p) = \ell$. Let $q \in (T(p) \cap Dom) \setminus \{p\}$. We have two cases:

q is a child of p: So, $\|p,q\| = 1$. By definition, $q \in Dom$ in γ. Moreover, as q is not the root, $q.\alpha = k$ in γ by definition of $\mathcal{CLR}(k)$. Then, by lemma 28, $p.\alpha \geq k$ in γ and we consider two subcases:

$p.\alpha = k$ in γ: Then, $|p.\alpha - k| = 0$ and the lemma holds.

$p.\alpha > k$ in γ: Then $p.\alpha = \texttt{MinATall}(p) + 1$ and $\texttt{MinATall}(p) = k$ (because $q.\alpha = k$). So, $p.\alpha = k + 1 \geq 1$ and the lemma holds.

q is not a child of p in γ: Then, there is a child y of p such that $q \in T(y) \cap Dom) \setminus \{y\}$ in γ (note that $lvl(y) = \ell + 1$).

Consider the three following cases:

- $p.\alpha < k$ *in* γ. In this case, $y.\alpha \neq k$ by Lemma 28. So, we consider the two following subcases:
  - $y.\alpha < k$ *in* γ. By the induction hypothesis, we have:

$$\begin{array}{ll} \|y,q\| & > |y.\alpha - k| \\ \|p,q\| & > |y.\alpha - k| + 1 \\ \|p,q\| & > |\texttt{MaxAShort}(p) - k| + 1 \\ \|p,q\| & > |\texttt{MaxAShort}(p) - (k+1)| \\ \|p,q\| & > |\texttt{MaxAShort}(p) + 1 - (k+2)| \\ \|p,q\| & > |p.\alpha - (k+2)| \\ \|p,q\| & > |p.\alpha - k| \end{array}$$

  - $y.\alpha > k$ *in* γ. Then:

$$\begin{array}{ll} \texttt{MaxAShort}(p) + \texttt{MinATall}(p) & > 2k - 2 \\ \texttt{MaxAShort}(p) - k + 1 & > k - 1 - \texttt{MinATall}(p) \\ p.\alpha - k & > k - 1 - \texttt{MinATall}(p) \\ |k - 1 - \texttt{MinATall}(p)| & > |p.\alpha - k| \\ |k - \texttt{MinATall}(p)| + 1 & > |p.\alpha - k| \\ |k - y.\alpha| + 1 & > |p.\alpha - k| \\ |y.\alpha - k| + 1 & > |p.\alpha - k| \\ \|y,q\| + 1 & > |p.\alpha - k| \quad \text{(by the ind. hyp.)} \\ \|p,q\| & > |p.\alpha - k| \end{array}$$

- $p.\alpha = k$ *in* γ. Then, $|p.\alpha - k| = 0$ and as every proper descendant of p is at least at distance 1 from p, the lemma trivially holds.
- $p.\alpha > k$ *in* γ. So, we consider the two following subcases:

– $y.\alpha < k$.

$$\begin{array}{ll}
\texttt{MaxAShort}(p) + \texttt{MinATall}(p) & \leq 2k-2 \\
\texttt{MaxAShort}(p) + \texttt{MinATall}(p) + 1 & \leq 2k-1 \\
\texttt{MaxAShort}(p) + p.\alpha & \leq 2k-1 \\
p.\alpha - k & \leq k - \texttt{MaxAShort}(p) - 1 \\
|p.\alpha - k| & \leq |k - \texttt{MaxAShort}(p) - 1| \\
|p.\alpha - k| & \leq |k - \texttt{MaxAShort}(p)| + 1 \\
|p.\alpha - k| & \leq |\texttt{MaxAShort}(p) - k| + 1 \\
|p.\alpha - k| & \leq |y.\alpha - k| + 1 \\
|p.\alpha - k| & \leq \|y,q\| + 1 \quad \text{(by the ind. hyp.)} \\
\|p,q\| & \geq |p.\alpha - k|
\end{array}$$

– $y.\alpha \geq k$. By the induction hypothesis, we have:

$$\begin{array}{ll}
\|y,q\| & \geq |y.\alpha - k| \\
\|p,q\| & \geq |y.\alpha - k| + 1 \\
\|p,q\| & \geq |\texttt{MinATall}(p) - k| + 1 \\
\|p,q\| & \geq |\texttt{MinATall}(p) + 1 - k| \\
\|p,q\| & \geq |p.\alpha - k|
\end{array}$$

□

Theorem 20 *The set Dom of clusterheads computed by $\mathcal{CLR}(k)$ is a minimum cardinality k-dominating set of T.*

(Figure 8.4 illustrates the proof.)

Proof. Consider the set Dom of clusterheads defined in some terminal configuration computed by $\mathcal{CLR}(k)$ in T. We proceed by contradiction: Assume that there exists a k-dominating set DS of T such that $|DS| < |Dom|$. Pick a process p of maximum level such that $T(p) \cap Dom$ contains more processes than $T(p) \cap DS$, *i.e.*:

- $|T(p) \cap Dom| > |T(p) \cap DS|$, and
- $|T(q) \cap Dom| \leq |T(q) \cap DS|$ for any proper descendant q of p in T.

This means, in particular, that $p \in Dom$ but $p \notin DS$. By definition of Dom, $p.\alpha \leq k$. By property 3.(a), there exists a sequence of processes p_0,p_1,...,p_a, for $a = p.\alpha$, such that:

- $p_a = p$,
- the parent of p_i in T is p_{i+1}, for all $0 \leq i < a$, and
- $p_i.\alpha = i$, for all $0 \leq i \leq a$.

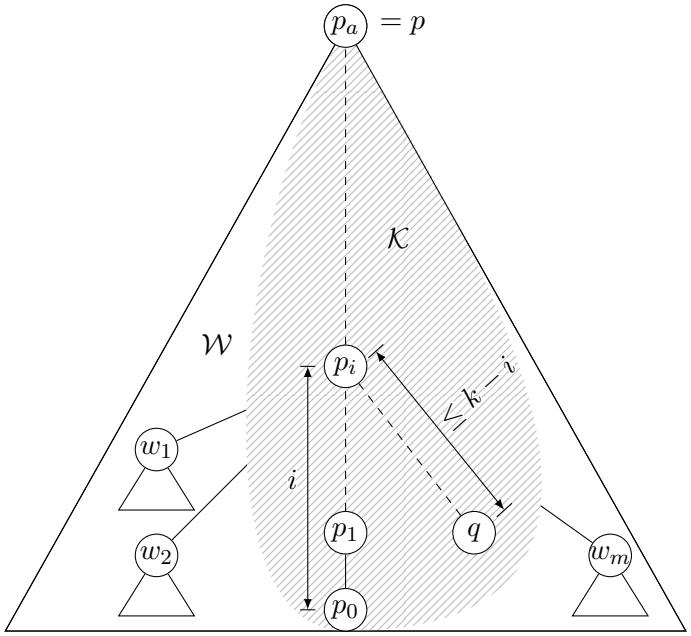


Figure 8.4 – Illustration of the proof of Theorem 20.

Let $\mathcal{K}$ be the set of all processes within k hops of p_0.

Claim I: $\mathcal{K}$ is a subset of $T(p)$.
Proof of Claim I: If p is the root of T, then the claim trivially holds. Otherwise, $a = p.\alpha = k$, which implies that p_0 is k hops below p, and thus the claim holds.

Claim II: $\mathcal{K} \cap Dom = \{p\}$.
Proof of Claim II: Suppose $q \in \mathcal{K}$ and $q \neq p$. Pick the process p_i that is closest to q. Then, q is at most $k - i$ (*i.e.*, $|p_i.\alpha - k|$) hops below p_i. By Lemma 29, $q \notin Dom$.

Let $\mathcal{W} = T(p) \setminus \mathcal{K}$. Then, $\mathcal{W}$ is the exact union of subtrees rooted at w_1, w_2,..., w_m, namely the processes not in $\mathcal{K}$ whose parents are in $\mathcal{K}$.

Each w_i is a proper descendant of p, and thus, by hypothesis, DS must have at least as many members as Dom in $\mathcal{W}$. Since DS has fewer members than Dom in $T(p)$, then DS must have fewer members than Dom in $\mathcal{K}$. By Claim II, $\mathcal{K} \cap DS = \emptyset$. This implies that DS contains no process within k hops of p_0, contradicting the hypothesis that DS is a k-dominating set. □

8.4 Competitiveness of k-Clustering

Unit-Disk Graphs. We now analyze the competitiveness, in terms of number of clusters, of $\mathcal{CLR}(k) \circ \mathcal{MIST} \circ \mathcal{BFST}$, in the special case that the network is a UDG in the plane, that is, the processes are fixed in the plane, and two processes can communicate if and only if their Euclidean distance in the plane is at most one. We first show, in Lemma 30, that the cardinality of the MIS computed by

$\mathcal{MIST} \circ \mathcal{BFST}$ is bounded by a multiple of the minimum cardinality of any k-clustering, then in Lemma 31, we show that the cardinality of Clr, the k-clustering built by $\mathcal{CLR}(k) \circ \mathcal{MIST} \circ \mathcal{BFST}$, is bounded by a multiple of that same minimum.

The proof of Lemma 30 makes use of the following result by Folkman and Graham [FG69].

Theorem 21 ([FG69]) *Let X be a compact convex region of the plane and $J \subseteq X$ such that the distance between any two distinct members of J is at least 1. Then, the cardinality of J is at most $\left\lfloor \frac{2}{\sqrt{3}}A(X) + \frac{1}{2}P(X) + 1 \right\rfloor$, where $A(X)$ and $P(X)$ are the area and the perimeter of X, respectively.*

Lemma 30 *For every connected UDG and every $k \geq 1$, any independent set I is of cardinality at most $\left(\frac{2\pi k^2}{\sqrt{3}} + \pi k + 1\right)$ times the cardinality of an optimum k-clustering.*

Proof. Consider any independent set I and any optimum k-clustering Opt of some UDG in the plane. Consider any clusterhead p in Opt and the surrounding disk of radius k centered at p in the plane. All processes that belongs to the k-cluster of p are within this disk. As the distance between any two distinct members of I is greater than 1, we can apply Theorem 21, that is, no more than $\left(\frac{2}{\sqrt{3}}(\pi k^2) + \frac{1}{2}(2\pi k) + 1\right)$ processes of I can be in this disk, thus in the k-cluster of p. By definition, every process belongs to a k-cluster. It follows that the cardinality of I is at most $\left(\frac{2\pi k^2}{\sqrt{3}} + \pi k + 1\right) \times |Opt|$. □

We now compare the maximal independent set computed by $\mathcal{MIST} \circ \mathcal{BFST}$ with the k-clustering set Clr computed by $\mathcal{CLR}(k) \circ \mathcal{MIST} \circ \mathcal{BFST}$.

Lemma 31 *For every connected network and every $k \geq 1$, let I be the MIS computed by $\mathcal{MIST} \circ \mathcal{BFST}$, the cardinality of Clr, the k-clustering built by $\mathcal{CLR}(k) \circ \mathcal{MIST} \circ \mathcal{BFST}$ is at most $1 + \frac{2}{k}(|I| - 1)$.*

Proof. By Lemma 25 (page 97), every k-cluster of Clr contains a path of $k+1$ processes (*i.e.*, of length k), except for the k-cluster which contains r. Since Clr is built on T_{MIS}, by Property 1 (page 44), this path contains $\lceil \frac{k}{2} \rceil$ processes of $I \setminus \{r\}$. Thus, $|Clr| - 1$ k-clusters of Clr contain at least $\lceil \frac{k}{2} \rceil$ processes of $I \setminus \{r\}$. We have:

$$\begin{array}{ll} (|Clr| - 1) \times \lceil \frac{k}{2} \rceil & \leq |I \setminus \{r\}| \\ (|Clr| - 1)\frac{k}{2} & \leq |I| - 1 \\ |Clr| - 1 & \leq \frac{2}{k}(|I| - 1) \\ |Clr| & \leq 1 + \frac{2}{k}(|I| - 1) \end{array}$$

□

By Lemmas 30 and 31, we deduce that $|Clr| \leq 1 - \frac{2}{k} + \left(\frac{4\pi k}{\sqrt{3}} + 2\pi + \frac{2}{k}\right)|Opt|$, and since $\frac{4\pi}{\sqrt{3}} \approx 7.2552$, we can claim:

Theorem 22 *For every connected UDG and every $k \geq 1$, $\mathcal{CLR}(k) \circ \mathcal{MIST} \circ \mathcal{BFST}$ computes a $7.2552k + O(1)$-approximation of the optimum k-clustering in terms of cardinality.*

Theorem 23 *For every connected approximate disk graph in the plane with approximation ratio λ, and every $k \geq 1$, $\mathcal{CLR}(k) \circ \mathcal{MIST} \circ \mathcal{BFST}$ computes a $7.2552\lambda^2 k + O(\lambda)$-approximation of the optimum k-clustering in terms of cardinality.*

Proof. As in the proof of Lemma 30, we make use of Theorem 21, but we then consider the surrounding disk of radius λk centered at any clusterhead of an optimum k-clustering *Opt*. It follows that no more than $\left(\frac{2}{\sqrt{3}}(\pi\lambda^2 k^2) + \frac{1}{2}(2\pi\lambda k) + 1\right)$ processes can be independent in this disk, and thus no more than that same number can be in any k-cluster of *Opt*. It follows that the cardinality of any independent set in an ADG is at most $\left(\frac{2\pi\lambda^2 k^2}{\sqrt{3}} + \pi\lambda k + 1\right)$ times the one of an optimum k-clustering *Opt*. By Lemma 31 and since $\frac{4\pi}{\sqrt{3}} \approx 7.2552$, we are done. □

Chapter 9

Experimentation

Contents

In this chapter, we evaluate both our k-dominating set and k-clustering algorithms, presented in preceding Chapters 7 and 8, beyond the worst case. We ran simulations in random topologies to study their average performance in terms of number of clusterheads. We first present the simulation setting in the next section. Then, in Section 9.2, we give our simulations results.

9.1 Simulation Method

We obtained our experimental results using an event-driven simulator for wireless sensor networks (WSNs), called *Sinalgo* [Flu08]. In this simulator, processes are randomly deployed on a square plane. Processes are motionless and equipped with radio. Two processes can communicate if and only if their Euclidean distance is at most R, where R is the transmission range. So, the network topology is a unit-disk graph (UDG).

We consider connected UDG networks of $n = 1000$ nodes deployed using a uniform random distribution on a $4000m$-side square, where m is an arbitrary unit of length. We tune the transmission range R to control the average degree $\overline{\delta}$ of the network, according that $\overline{\delta} = \frac{n}{\text{area}}\pi R^2$ where area $= 4000^2$ here. We make varying $\overline{\delta}$ from 10 to 30 and k from 1 to 5. For each setting, the average number of clusterheads is computed over 50 connected UDGs, randomly generated once.

We only present here the results obtained with $k = 5$. The general trends observed for $k = 5$ are representative: we have also observed them in other cases.

9.2 Results Analysis

Now, we first consider the simplest versions of our algorithms to construct a set of clusterheads of bounded size. Then, we put the minimization algorithm given in [DDL09] in the simulation loop.

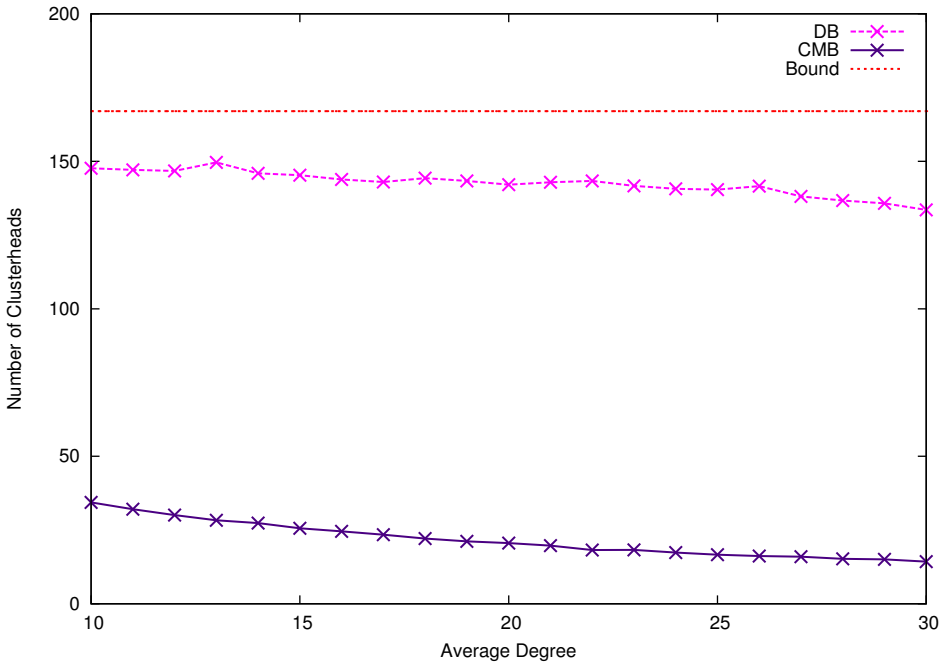


Figure 9.1 – DB vs. CMB vs. the theoretical bound, for $n = 1000$, $k = 5$, and a square field of size $4000m$.

9.2.1 k-Clustering of Bounded Size

In Chapter 7, the two first layers $\mathcal{ST}$ and $\mathcal{DS}(k)$ of our algorithm $\mathcal{SMDS}(k)$, can be composed to construct a k-dominating set of the network. The first layer $\mathcal{ST}$ constructs a spanning tree of the network. We considered three different constructions to implement it: a breadth-first search tree (BFS tree) [HC92], a depth-first spanning tree (DFS tree) [CD94], and an arbitrary spanning tree [CYH91]. We tested each of them with the second layer $\mathcal{DS}(k)$ and selected the BFS tree which had the best impact on the size of the output k-dominating set. Here, we implemented $\mathcal{ST}$ using the breadth-first search tree (BFS tree) construction algorithm given in [HC92] and denote it by $\mathcal{BFST}$. Here, we denote $\mathcal{DS}(k) \circ \mathcal{BFST}$ by DB.

In Chapter 8, our three-layered algorithm $\mathcal{CLR}(k) \circ \mathcal{MIST} \circ \mathcal{BFST}$, denoted by CMB in the following, constructs a k-clustering of the network.

Note that both DB and CMB algorithms compute a set of clusterheads which is not necessarily minimal. However, its size is bounded by $\lceil \frac{n}{k+1} \rceil$, as proven in Theorem 12 (page 82) and Theorem 18 (page 97).

We compared the average performance of both DB and CMB against the aforementioned theoretical bound. The experimental results are given in Figure 9.1. They confirm that CMB is well-suited for WSNs, since its average performances are drastically better than the theoretical bound, which holds for all arbitrary connected graphs. Besides, they show that DB poorly performs, as its average performances are close to the theoretical upper bound representing the worst case.

Algorithm	Memory Requirement	Round Complexity	Bound	Minimal
DB	$O(\log k + \log n + k \log \frac{N}{k})$	$O(n)$	$\lceil \frac{n}{k+1} \rceil$	No
CMB	$O(\log k + \log n)$	$O(n)$	$\lceil \frac{n}{k+1} \rceil$	No
Min	$O(\log k + \log n)$	$O(n)$	$\max(1, n/\lceil \frac{k+1}{2} \rceil)$	Yes
MinDB	$O(\log k + \log n + k \log \frac{N}{k})$	$O(n)$	$\lceil \frac{n}{k+1} \rceil$	Yes
MinCMB	$O(\log k + \log n)$	$O(n)$	$\lceil \frac{n}{k+1} \rceil$	Yes

Table 9.1 – Features of each algorithm.

Note also that, the number of clusterheads decreases when the average degree increases because the diameter of the network also decreases in that case. This trend can be also observed in all other curves.

9.2.2 Minimal k-Clustering

We also implemented the minimization algorithm $\mathcal{MIN}(k)$, already mentioned in Chapter 7, and actually given in [DDL09].

This algorithm can be used without input, and in this case, it builds an unconstrained minimal k-dominating set of the network, whose size is at most $\max(1, n/\lceil \frac{k+1}{2} \rceil)$, as shown in [DDL09, page 153]. In the following, we denote the standalone version of this algorithm by Min.

Otherwise, $\mathcal{MIN}(k)$ can be used with an input k-dominating set, and in this case, it computes a minimal k-dominating set which is a subset of the input k-dominating set. So, we can use this minimization module to implement the third layer of our algorithm $\mathcal{SMDS}(k)$ given in Chapter 7. Here, we denote its full implementation $\mathcal{MIN}(k) \circ \mathcal{DS}(k) \circ \mathcal{BFST}$ by MinDB. Finally, we can also use the minimization module as a fourth layer to our k-clustering algorithm presented in Chapter 8. Here, we denote $\mathcal{MIN}(k) \circ \mathcal{CLR}(k) \circ \mathcal{MIST} \circ \mathcal{BFST}$ by MinCMB.

We recall the main features of each algorithm in Table 9.1. To the best of our knowledge, these are the only self-stabilizing algorithms that guarantee a bound on the number of clusterheads.

We compared the two versions of our k-dominating set algorithm, DB and MinDB to see if the minimization module really impacts the result. As we can see in Figure 9.2, it drastically reduced the size of the computed k-dominating sets. The average ratio between the two performance curves is higher than 18.

Next, we compared our k-clustering algorithms CMB and MinCMB to observe the impact of the minimization module on the size of the computed set of clusterheads. As shown in Figure 9.3, the minimization reduced the size of the computed k-dominating sets by a factor just under 3.

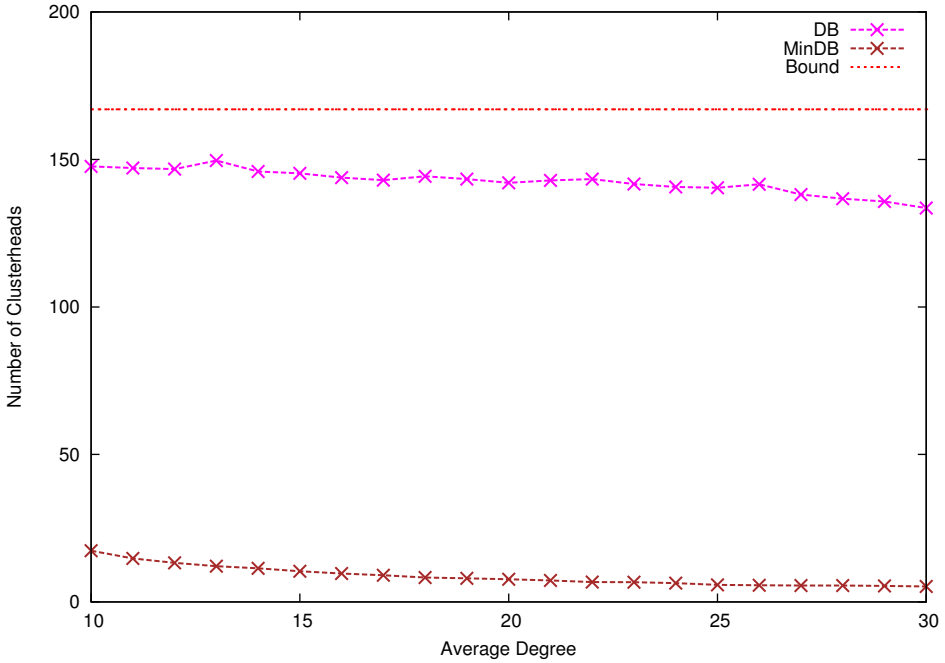


Figure 9.2 – DB vs. MinDB, for $n = 1000$, $k = 5$, and a square field of size $4000m$.

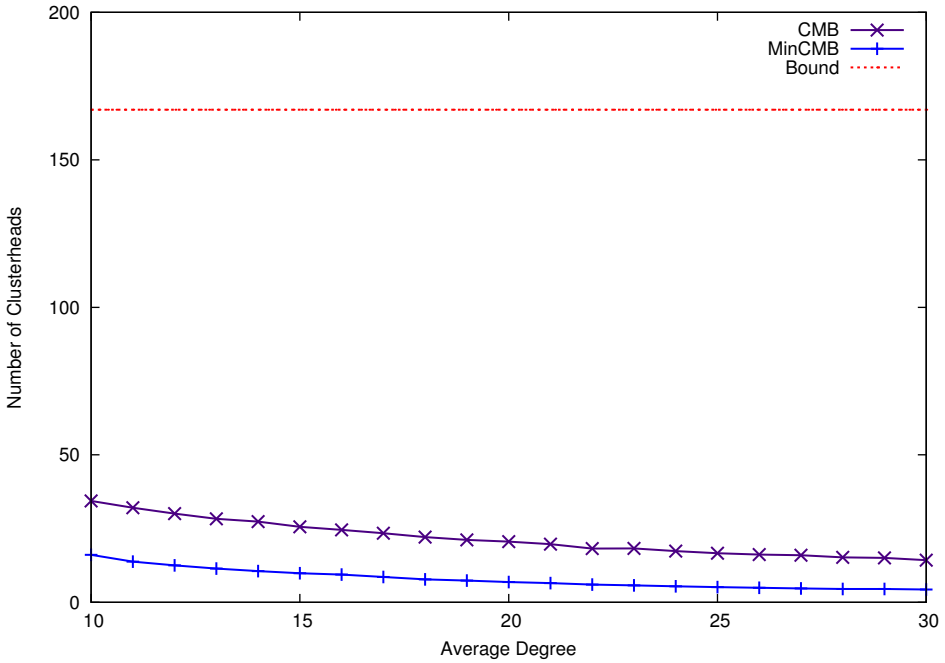


Figure 9.3 – CMB vs. MinCMB, for $n = 1000$, $k = 5$, and a square field of size $4000m$.

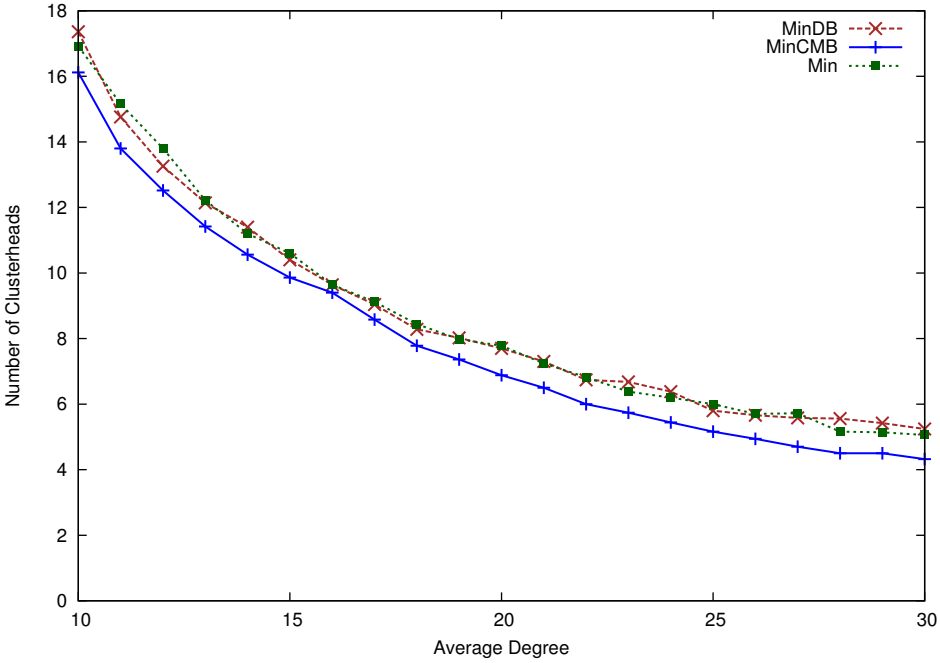


Figure 9.4 – MinDB vs. MinCMB vs. Min, for $n = 1000$, $k = 5$, and a square field of size $4000m$.

Finally, we compared both algorithms MinDB and MinCMB to the standalone algorithm Min given in [DDL09]. Figure 9.4 presents the experimental results, using a tighter scale for the vertical axis than in previous figures. We can remark that results are really close, but still algorithm MinCMB offers the best performances.

Part IV

(f, g)-Alliance

Chapter 10

(f, g)-Alliance with Safe Convergence

Contents

In this chapter, we consider the (f, g)-*alliance problem* which consists in constructing an (f, g)-alliance of the network, as defined hereafter. The (f, g)-alliance is a generalization of several spanning structures that are of interest in distributed computing. Here we give a silent self-stabilizing algorithm with safe convergence to compute a *minimal* (f, g)-*alliance* under an unfair daemon.

In next section, we define an (f, g)-alliance, exhibit relations with other spanning structures, and set the context of our contribution. Then, we give our algorithm $\mathcal{MA}(f, g)$ in Section 10.2. In Section 10.3, we prove the correctness of $\mathcal{MA}(f, g)$ and analyze its complexity analysis.

10.1 Introduction

10.1.1 Definition of (f, g)-Alliance

Let $G = (V, E)$ be an undirected graph and f, g two functions mapping processes to non-negative integers. A subset of processes $A \subseteq V$ is an (f, g)-*alliance* of G if and only if

$$(\forall p \in V \setminus A, |\mathcal{N}(p) \cap A| \geq f(p)) \wedge (\forall p \in A, |\mathcal{N}(p) \cap A| \geq g(p))$$

Moreover,

- A is *minimal* if and only if no proper subset of A is an (f, g)-alliance of G;
- A is 1-*minimal* if and only if $\forall p \in A$, $A \setminus \{p\}$ is not an (f, g)-alliance.

Surprisingly, a 1-*minimal* (f, g)-alliance is not necessarily a *minimal* (f, g)-alliance [DPRS11]. However, we have the following property:

Property 4 [DPRS11] *Given two functions f and g mapping processes to non-negative integers, we have:*

1. *Every minimal (f, g)-alliance is a 1-minimal (f, g)-alliance, and*
2. *if $f \geq g$, every 1-minimal (f, g)-alliance is a minimal (f, g)-alliance.*

10.1.2 Relationship with other Data Structures

The (f, g)-alliance is a generalization of several spanning structures that are of interest in distributed computing. Consider any subset S of processes:

1. S is a (minimal) dominating set if and only if S is a (minimal) $(1, 0)$-alliance;
2. more generally, S is a (minimal) k-redundant dominating set if and only if S is a (minimal) $(k, 0)$-alliance;
3. S is a (minimal) k-tuple dominating set if and only if S is a (minimal) $(k, k-1)$-alliance;
4. S is a (minimal) global defensive alliance if and only if S is a (minimal) $(f, 0)$-alliance, such that $\forall p \in V$, $f(p) = \left\lceil \frac{\delta(p)}{2} \right\rceil$;
5. S is a (minimal) global offensive alliance if and only if S is a (minimal) $(1, g)$-alliance, such that $\forall p \in V$, $g(p) = \left\lceil \frac{\delta(p)}{2} \right\rceil$.

Note that (f, g)-alliances also have applications in the field of population protocols [AAER07], or server allocation in computer networks [GMOR05].

10.1.3 Our Contribution

We give a silent self-stabilizing algorithm, $\mathcal{MA}(f,g)$, that computes a minimal (f,g)-alliance in any undirected network, under an unfair daemon, where f and g are integer-valued functions on processes, such that $f(p) \geq g(p)$ and $\delta(p) \geq g(p)$ for all p. Note that we assume that $\delta(p) \geq g(p)$ to ensure that an (f,g)-alliance always exists.

Given two functions f, g mapping processes to non-negative integers, we say $f \geq g$ if and only if $\forall p \in V, f(p) \geq g(p)$. We remark that the class of minimal (f,g)-alliances with $f \geq g$ generalizes the classes of minimal dominating sets, k-redundant dominating sets, k-tuple dominating sets, and global defensive alliance problems. However, minimal global offensive alliances do not belong to this class.

Our algorithm $\mathcal{MA}(f,g)$ is *safely converging* in the sense that starting from any configuration, it first converges to a (not necessarily minimal) (f,g)-alliance in at most four rounds, and then continues to converge to a minimal one in at most $5n+4$ additional rounds, where n is the size of the network. $\mathcal{MA}(f,g)$ uses $O(\log n)$ bits per process, and stabilizes to a terminal (legitimate) configuration in $O(\Delta^3 n)$ steps, where Δ is the degree of the network.

10.1.4 Related Work

The (f,g)-alliance problem is introduced in [DPRS11]. In the same paper, the authors give several distributed algorithms for that problem and its variants, but none of them is self-stabilizing. To the best of our knowledge, this has been the only publication on (f,g)-alliances up to now.

However, there have been results on particular instances of (minimal) (f,g)-alliances, *e.g.*, [KM06, SX07, Tur07, WWTZ12]. All of these consider arbitrary identified networks; however a safely converging solution is given only in [KM06]. Srimani and Xu [SX07] give a self-stabilizing algorithm to compute a minimal global defensive alliance in $O(n^3)$ steps; however, they assume a central daemon. Turau [Tur07] gives a self-stabilizing algorithm to compute a minimal dominating set in $9n$ steps, assuming an unfair (distributed) daemon. Wang *et al* [WWTZ12] give a self-stabilizing algorithm to compute a minimal k-redundant dominating set in $O(n^2)$ steps, assuming a central daemon. A safely converging self-stabilizing algorithm is given in [KM06] for computing a minimal dominating set. The algorithm first computes a (not necessarily minimal) dominating set in $O(1)$ rounds and then safely stabilizes to a *minimal* dominating set in $O(\mathcal{D})$ rounds, where $\mathcal{D}$ is the diameter of the network. However, they assume a synchronous daemon.

10.2 Algorithm $\mathcal{MA}(f, g)$

The formal code of $\mathcal{MA}(f, g)$ is given as Algorithm 6. Given input functions f and g, $\mathcal{MA}(f, g)$ computes a single Boolean variable $p.in\_\, a$ for each process p. For any configuration γ, let $A_\gamma = \{p \in V : p.in\_\, a\}$. If γ is terminal, then A_γ is a 1-*minimal* (f, g)-*alliance*, and consequently, if $f \geq g$, A_γ is a *minimal* (f, g)-*alliance*.

During an execution, a process may need to leave or join A. The basic idea of safe convergence is that it should be more difficult for a process to leave A than to join it. This permits quick recovery to a configuration in which A is an (f, g)-alliance, but not necessarily a minimal one.

10.2.1 Leaving A

Action `Leave` allows a process to leave A. To obtain 1-minimality, we allow a process p to leave A if

Requirement 1: p will have enough neighbors in A (*i.e.*, at least $f(p)$) once it has left, and

Requirement 2: each $q \in \mathcal{N}(p)$ will still have enough neighbors in A (*i.e.*, at least $g(q)$ or $f(q)$, depending on whether q is in A) once p has been deleted from A.

Ensuring Requirement 1. To maintain Requirement 1, we implement our algorithm in such a way that deletion from A is *locally sequential*, *i.e.*, during a step, at most one process can leave A in the neighborhood of each process p (including p itself). Using this locally sequential mechanism, if a process p wants to leave A, it must first verify that $\texttt{NbA}(p) = |\{q \in \mathcal{N}(p) : q.in\_\, a\}|$ is greater or equal to $f(p)$ before leaving A. Hence, if p actually leaves A, it is the only one in its neighborhood allowed to do so; consequently, Requirement 1 still holds once p has left A.

The locally sequential mechanism is implemented using a neighbor pointer $p.choice$ at each process p, which takes value in $\mathcal{N}(p) \cup \{\perp\}$; $p.choice = q \in \mathcal{N}(p)$ means that p authorizes q to leave A, while $p.choice = \perp$ means that p does not authorize any neighbor to leave A. The value of $p.choice$ is maintained using Action `Vote`, which will be defined later.

To leave A, a process p should not authorize any neighbor to leave A ($p.choice = \perp$) and should be authorized to leave by all of its neighbors ($\forall q \in \mathcal{N}(p),\ q.choice = p$). For example, consider the $(1, 0)$-alliance in Figure 10.1. Only Process 2 is able to leave A. Process 2 can leave A because it has enough neighbors in A (*i.e.*, 2 neighbors, while $f(2) = 1$); if Process 2 leaves A, it will still have two neighbors in A, and Requirement 1 will not be violated.

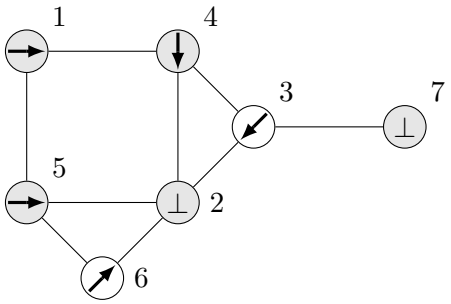


Figure 10.1 – Neighbor pointers when computing a minimal $(1, 0)$-alliance. Numbers indicate UIDs. A is the set of gray processes. The value of *choice* is represented by an arrow or a tag "$\perp$" inside the process.

Ensuring Requirement 2. Requirement is also maintained by the fact that a process p must have authorization from each of its neighbors to leave A. A neighbor q can give such an authorization to p only if q still has enough neighbors in A without p. For a process q to authorize a neighbor p to leave A, p must currently be in A, *i.e.*, $p.in\_\, a = \text{TRUE}$, and q must have more neighbors than necessary in A, *i.e.*, the predicate `HasExtra`(q) should be true, meaning that $\mathcal{N}(q) \cap A$ has more than $g(q)$, respectively $f(q)$, members if q is in A, respectively not in A. For example, consider the $(1, 0)$-alliance in Figure 10.1. Processes 4 and 5 can designate Process 2 because they belong to A and $g(4) = g(5) = 0$. Moreover, Processes 3 and 6 can designate Process 2 because they do not belong to A and $f(3) = f(6) = 1$: if Process 2 leaves A, Process 3 (resp. Process 6) still has one neighbor in A, which is Process 7 (resp. Process 5).

Busy Processes. It is possible that a neighbor p of q cannot leave A — in this case p is said to be *busy* — because one of these two conditions is TRUE:

(i) **`NbA`$(p) < f(p)$:** in this case, p does not have enough neighbors in A to be allowed to leave A.

(ii) **$\neg$`IsExtra`(p):** in this case, at least one neighbor of p needs p to stay in A.

If q chooses such a neighbor p, this may lead to a deadlock. We use the Boolean variable $p.busy$ to inform q that one of the two aforementioned conditions holds for p. Action `Flag` maintains $p.busy$. So, to prevent deadlock, q must not choose any neighbor p for which $p.busy = \text{TRUE}$.

A process p evaluates Condition (i) by reading the variables $in\_\, a$ of all its neighbors. On the other hand, evaluation of Condition (ii) requires that p knows, for each of its neighbors, both its status ($in\_\, a$) and the number of its own neighbors that are in A. This latter information is obtained using an additional variable, $nb\_\, a$, where each process maintains, using Action `Count`, the number of its neighbors that are in A.

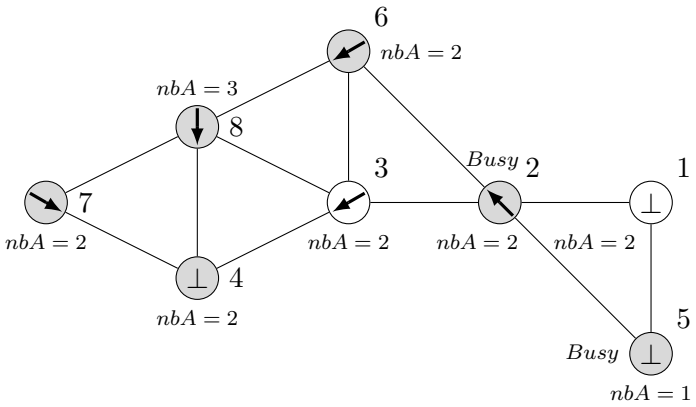


Figure 10.2 – Busy processes when computing a minimal $(2, 0)$-alliance. Values of nb_a are also given.

In Figure 10.2, consider the $(2, 0)$-alliance. Process 5 is busy because of Condition (i): it has only one neighbor in A, while $f(5) = 2$. Process 2 is busy because of Condition (ii): its neighbor 1 is not in A, $f(1) = 2$, and has only two neighbors in A, so it cannot authorize any of its neighbors to leave. Consequently, Process 1 cannot designate any neighbor (all its neighbors in A are busy); while Process 3 should not designate Process 2.

Action `Vote`. Hence, the value of $p.choice$ is chosen, using Action `Vote`, as follows:

1. $p.choice$ is set to $\perp$ if the condition $\texttt{Cand}(p) \neq \emptyset \wedge \texttt{HasExtra}(p) \wedge (\texttt{IamCand}(p) \Rightarrow \texttt{MinCand}(p) < p)$ in Macro $\texttt{ChosenCand}(p)$ is FALSE, *i.e.*, if one of the following conditions holds:

   - $\texttt{Cand}(p) = \emptyset$, which means that no neighbor of p can leave A.
   - $\texttt{HasExtra}(p) =$ FALSE, which means that p cannot authorize any neighbor to leave A.
   - $\texttt{IamCand}(p) \wedge p < \texttt{MinCand}(p)$, which means that p is also candidate to leave A and has higher priority to leave A than any other candidate in its neighborhood. (Remember that to be allowed to leave A, p should, in particular, satisfy $p.choice = \perp$.)

     The aforementioned priorities are based on process UIDs, *i.e.*, for every two process u and v, u has higher priority than v if and only if the UID of u is smaller than the UID of v.

2. Otherwise, p uses $p.choice$ to designate a neighbor that is in A, and not busy, in order to authorize it to leave A. If p has several possible candidates among its neighbors, it selects the one of highest priority (*i.e.*, of smallest UID). For example, if we consider the $(2, 0)$-alliance in Figure 10.2, then we can see that Process 3 designates Process 4 because it is its smallest neighbor that is both in A and not busy.

There is one last problem: A process q may change its pointer while simultaneously one of its neighbors p leaves A, and consequently Requirement 2 may be violated. Indeed, q chooses new candidate assuming that p remains in A. This may happen only if the previous value of $q.choice$ was p. To avoid this situation, we do not allow q to directly change $q.choice$ from one neighbor to another. Each time q wants to change its pointer, if $q.choice \in \mathcal{N}(q)$, q first resets $q.choice$ to $\perp$; see `Choice`(q).

Figures 10.3 and 10.4 illustrate this last issue in the case of a $(1, 0)$-alliance. In the step from Configuration (a) to Configuration (b) of Figure 10.3, Process 2 directly changes its pointer from 3 to 1. Simultaneously, 3 leaves A. So, Process 2 authorizes Process 1 to leave A, while it should not do so. After that, Process 1 is authorized to leave A and does so at the step from Configuration (b) to Configuration (c), and thus Requirement 2 is violated. Figure 10.4 illustrates how we solve the problem. In Configuration (b), Process 3 has left, but the pointer of Process 2 is equal to $\perp$. So, Process 1 cannot leave yet, and Process 2 will not authorize it to leave.

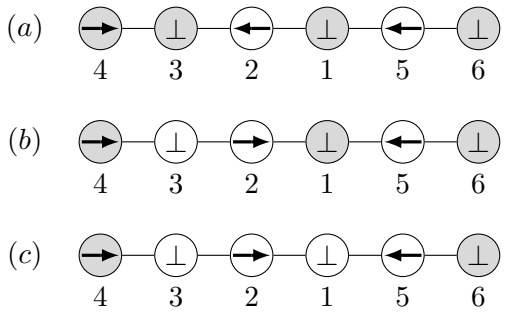


Figure 10.3 – Requirement 2 violation when computing a minimal $(1, 0)$-alliance. (We only show the values that are needed in the discussion.)

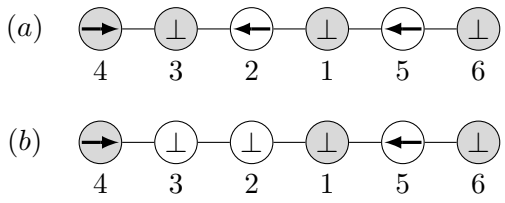


Figure 10.4 – The reset of the neighbor pointer is applied to the example of Figure 10.3.

Algorithm 6 $\mathcal{MA}(f, g)$, code for each process p

Variables:

$p.in\_a$: Boolean
$p.busy$: Boolean
$p.choice \in \mathcal{N}(p) \cup \{\perp\}$
$p.nb\_a \in [0..\delta(p)]$

Macros:

$\texttt{NbA}(p) = |\{q \in \mathcal{N}(p), q.in\_a\}|$
$\texttt{Cand}(p) = \{q \in \mathcal{N}(p), q.in\_a \wedge \neg q.busy\}$
$\texttt{MinCand}(p) = \min(\texttt{Cand}(p) \cup \{\infty\})$
$\texttt{ChosenCand}(p) =$ **if** $\texttt{Cand}(p) \neq \emptyset \wedge \texttt{HasExtra}(p) \wedge (\texttt{IamCand}(p) \Rightarrow \texttt{MinCand}(p) < p)$
then $\texttt{MinCand}(p)$
else $\perp$
$\texttt{Choice}(p) =$ **if** $p.choice = \perp$
then $\texttt{ChosenCand}(p)$
else $\perp$

Predicates:

$\texttt{IsMissing}(p) \equiv \exists q \in \mathcal{N}(p), (\neg q.in\_a \wedge q.nb\_a < f(q)) \vee (q.in\_a \wedge q.nb\_a < g(q))$
$\texttt{IsExtra}(p) \equiv \forall q \in \mathcal{N}(p), (\neg q.in\_a \Rightarrow q.nb\_a > f(q)) \wedge (q.in\_a \Rightarrow q.nb\_a > g(q))$
$\texttt{HasExtra}(p) \equiv (\neg p.in\_a \Rightarrow \texttt{NbA}(p) > f(p)) \wedge (p.in\_a \Rightarrow \texttt{NbA}(p) > g(p))$
$\texttt{IsBusy}(p) \equiv \texttt{NbA}(p) < f(p) \vee \neg\texttt{IsExtra}(p)$
$\texttt{IamCand}(p) \equiv p.in\_a \wedge \neg\texttt{IsBusy}(p)$
$\texttt{MustJoin}(p) \equiv \neg p.in\_a \wedge (\texttt{NbA}(p) < f(p) \vee \texttt{IsMissing}(p)) \wedge (\forall q \in \mathcal{N}(p), q.choice \neq p)$
$\texttt{CanLeave}(p) \equiv p.in\_a \wedge \texttt{NbA}(p) \geq f(p) \wedge (\forall q \in \mathcal{N}(p), q.choice = p) \wedge p.choice = \perp$

Actions:

Join	::	$\texttt{MustJoin}(p)$	$\rightarrow$	$p.in\_a \leftarrow$ TRUE; $p.choice \leftarrow \perp$; $p.nb\_a \leftarrow \texttt{NbA}(p)$
Vote	::	$p.choice \neq \texttt{ChosenCand}(p)$	$\rightarrow$	$p.choice \leftarrow \texttt{Choice}(p)$; $p.nb\_a \leftarrow \texttt{NbA}(p)$; $p.busy \leftarrow \texttt{IsBusy}(p)$
Count	::	$p.nb\_a \neq \texttt{NbA}(p)$	$\rightarrow$	$p.nb\_a \leftarrow \texttt{NbA}(p)$
Flag	::	$p.busy \neq \texttt{IsBusy}(p)$	$\rightarrow$	$p.busy \leftarrow \texttt{IsBusy}(p)$
Leave	::	$\texttt{CanLeave}(p)$	$\rightarrow$	$p.in\_a \leftarrow$ FALSE

10.2.2 Joining A

Action `Join` allows a process to join A. A process p not in A must join A if:

(1) p does not have enough neighbors in A ($\texttt{NbA}(p) < f(p)$), or

(2) a neighbor of p needs p to join A ($\texttt{IsMissing}(p)$).

Moreover, to prevent p from cycling in and out of A, we require that every neighbor of p stop designating it (with their *choice* pointer) before p can join A (again). Note that all neighbors of p stop designating p immediately after it leaves A; see Action `Vote`. (Actually, this introduces a delay of only one round.)

A process evaluates condition (1) by reading the variables $in\_\ a$ of all its neighbors. To evaluate condition (2), it needs to know, for each neighbor q, both its status w.r.t. A ($q.in\_\ a$) and the number of its neighbors that are in A ($q.nb\_\ a$).

10.3 Correctness and Complexity Analysis

Recall that in any configuration γ, we define the set $A_\gamma = \{p \in V, p.in\_\ a\}$. (We omit the subscript γ when it is clear from the context.) In the next subsection, we define some predicates. Subsection 10.3.2 is dedicated to the proof of self-stabilization of $\mathcal{MA}(f,g)$ assuming an unfair daemon. We study the safe convergence of $\mathcal{MA}(f,g)$ in Subsection 10.3.3.

10.3.1 Predicates

First, throughout the section, we will use the notion of a *closed predicate*: Let P be a predicate over configuration of $\mathcal{MA}(f,g)$. P is *closed* if and only if $\forall \gamma, \gamma' \in \mathcal{C}$, $P(\gamma) \wedge \gamma \mapsto \gamma' \Rightarrow P(\gamma')$.

Let now define some predicates. First, for every process p,

$$\texttt{Fga}(p) \stackrel{\texttt{def}}{=} (\neg p.in\_\ a \Rightarrow \texttt{NbA}(p) \geq f(p)) \wedge (p.in\_\ a \Rightarrow \texttt{NbA}(p) \geq g(p))$$

When a process p satisfies $\texttt{Fga}(p)$, this means that it is locally correct, *i.e.*, it has enough neighbors in A according to its status. Then, by definition we have:

Remark 5 *A is an (f,g)-alliance if and only if $\forall p \in V$, $\texttt{Fga}(p)$.*

For every process p,

$$\texttt{NbAOk}(p) \stackrel{\texttt{def}}{=} (\neg p.in\_\ a \Rightarrow p.nb\_\ a \geq f(p)) \wedge (p.in\_\ a \Rightarrow p.nb\_\ a \geq g(p))$$

This predicate is always used in conjunction with $\texttt{Fga}(p)$. When both predicates are TRUE at p, this means that p is locally correct and the variable $p.nb\_\ a$ gives this information to the neighbors of p.

For every process p,

$$\texttt{ChoiceOk}(p) \stackrel{\texttt{def}}{=} (p.choice \neq \bot \wedge p.choice.in\_a) \Rightarrow \texttt{HasExtra}(p)$$

Once $\texttt{ChoiceOk}(p)$ holds at p, no neighbor of p can make p locally incorrect by leaving A.

The following predicates are defined over configurations of $\mathcal{MA}(f,g)$:

$$Spec_{1-Minimal} \stackrel{\texttt{def}}{=} A \text{ is a 1-minimal } (f,g)\text{-alliance}$$
$$Spec_{Minimal} \stackrel{\texttt{def}}{=} A \text{ is a minimal } (f,g)\text{-alliance}$$

10.3.2 Self-Stabilization

10.3.2.1 Partial Correctness

We now show that in any terminal configuration γ, the specification of $\mathcal{MA}(f,g)$ is achieved. To see this, we first show that A is an (f,g)-alliance in γ (Lemma 33), then we show that A is 1-minimal in γ, so if $f \geq g$, A is also a minimal (f,g)-alliance (Lemma 34). To show these two results, we use two intermediate claims: Lemma 32 and Corollary 3. The former states that every process of A is busy in γ, meaning that either p has not enough neighbors in A to leave A, or at least one neighbor of p requires that p stays in A, *i.e.,* A is 1-minimal. The latter is a simple corollary of Lemma 32 and states that no process authorizes a neighbor to leave A in γ.

In any terminal configuration, Action `Count` is disabled at every process, so:

Remark 6 *In any terminal configuration of $\mathcal{MA}(f,g)$, for every process p,* $p.\text{nb\_a} = \texttt{NbA}(p) = |\{q \in \mathcal{N}(p), q.\text{in\_a}\}|$.

Lemma 32 *In any terminal configuration of $\mathcal{MA}(f,g)$, for every process p,* $p.\text{in\_a} \Rightarrow p.\text{busy}$.

Proof. By contradiction. Let γ be a terminal configuration of $\mathcal{MA}(f,g)$ and assume that there is at least one process p such that $p.in\_a = \text{TRUE}$ and $p.busy = \text{FALSE}$ in γ. Then, for each such process p, we have $\texttt{IsBusy}(p) = \text{FALSE}$ in γ, because Action `Flag` is disabled at every process.

Let

$$p_{\min} = \min\{p \in V, p.in\_a = \text{TRUE} \wedge p.busy = \text{FALSE}\} \text{ in } \gamma \qquad (1)$$

Since $\neg\texttt{IsBusy}(p_{\min})$ in γ, we also have:

$\texttt{IsExtra}(p_{\min})$
$\forall q \in \mathcal{N}(p_{\min}), (\neg q.in\_a \Rightarrow q.nb\_a > f(q)) \wedge (q.in\_a \Rightarrow q.nb\_a > g(q))$
$\forall q \in \mathcal{N}(p_{\min}), (\neg q.in\_a \Rightarrow \texttt{NbA}(q) > f(q)) \wedge (q.in\_a \Rightarrow \texttt{NbA}(q) > g(q))$ by Remark 6
$\forall q \in \mathcal{N}(p_{\min}), \texttt{HasExtra}(q)$ (2)

Then, because $p_{\min}.in\_\, a = \text{TRUE} \wedge p_{\min}.busy = \text{FALSE}$ in γ we have:

$$\forall q \in \mathcal{N}(p_{\min}), p_{\min} \in \texttt{Cand}(q) \quad (3)$$
$$\forall q \in \mathcal{N}(p_{\min}), \texttt{Cand}(q) \neq \emptyset \quad (4)$$

By (1) and (3), in γ we have:

$$\forall q \in \mathcal{N}(p_{\min}), \texttt{MinCand}(q) = p_{\min} \quad (5)$$

By (1) and (5), in γ we have:

$$\forall q \in \mathcal{N}(p_{\min}), (\texttt{IamCand}(q) \Rightarrow \texttt{MinCand}(q) < q) \quad (6)$$

By (2), (4), (5), (6) and the fact that Action `Vote` is disabled, in γ we have:

$$\forall q \in \mathcal{N}(p_{\min}), \texttt{ChosenCand}(q) = p_{\min}$$
$$\forall q \in \mathcal{N}(p_{\min}), q.choice = p_{\min} \quad (7)$$

By definition, $\texttt{IamCand}(p_{\min})$ holds in γ. Moreover, by (1), $\texttt{MinCand}(p_{\min}) > p_{\min}$ in γ. So, $\texttt{MinCand}(p_{\min}) < p_{\min}$ is FALSE in γ. Hence, in γ we have $(\texttt{IamCand}(p_{\min}) \Rightarrow \texttt{MinCand}(p_{\min}) < p_{\min}) = \text{FALSE}$, and consequently:

$$\texttt{ChosenCand}(p_{\min}) = \bot$$
$$p_{\min}.choice = \bot \quad \text{(Action } \texttt{Vote} \text{ is disabled)} \quad (8)$$

Finally, because $\neg\texttt{IsBusy}(p_{\min})$ holds in γ, we have $\texttt{NbA}(p_{\min}) \geq f(p_{\min})$ in γ. So, by (7), (8), and the fact that $p_{\min}.in\_\, a = \text{TRUE}$ in γ, we can conclude that $\texttt{CanLeave}(p_{\min})$ holds in γ, that is, $p_{\min}$ is enabled in γ, contradiction. □

By Lemma 32, for every process p, $\texttt{Cand}(p) = \emptyset$ in any terminal configuration γ. Thus $\texttt{ChosenCand}(p) = \bot$ in γ, and from the negation of the guard of Action `Vote`, we have:

Corollary 3 *In any terminal configuration of $\mathcal{MA}(f, g)$, for every process p, p.*choice $= \bot$.

Lemma 33 *In any terminal configuration of $\mathcal{MA}(f, g)$, A is an (f, g)-alliance.*

Proof. Let γ be a terminal configuration. By Remark 5, we merely need show that every process p satisfies $\texttt{Fga}(p)$ in γ. Consider the following two cases:

***p* ∉ *A* in *γ*:** First, by definition, $p.in\_\, a = \text{FALSE}$ in γ. Then, γ being terminal, $\neg\texttt{MustJoin}(p)$ holds in γ. $\neg\texttt{MustJoin}(p) = \neg(\neg p.in\_\, a \wedge (\texttt{NbA}(p) < f(p) \vee \texttt{IsMissing}(p)) \wedge (\forall q \in \mathcal{N}(p), q.choice \neq p)) = p.in\_\, a \vee (\texttt{NbA}(p) \geq f(p) \wedge \neg\texttt{IsMissing}(p)) \vee (\exists q \in \mathcal{N}(p), q.choice = p)$. By $p.in\_\, a = \text{FALSE}$ and Corollary 3, $\neg\texttt{MustJoin}(p)$ in γ implies that $\texttt{NbA}(p) \geq f(p) \wedge \neg\texttt{IsMissing}(p)$ in γ. So, $\neg p.in\_\, a \wedge \texttt{NbA}(p) \geq f(p)$ holds in γ, which implies that $\texttt{Fga}(p)$ holds in γ.

***p* ∈ *A* in *γ*:** First, by definition, $p.in\_\ a$ = TRUE in γ. We need to show that $\texttt{Fga}(p)$ = TRUE in γ. Assume $\texttt{Fga}(p)$ = FALSE. Then, $\texttt{NbA}(p) < g(p)$. As $\delta(p) \geq g(p)$, $\exists q \in \mathcal{N}(p)$, $\neg q.in\_\ a$ in γ. By Remark 6, $p.nb\_\ a < g(p)$ in γ. So, as $p \in \mathcal{N}(q)$, $\texttt{IsMissing}(q)$ holds in γ. Now, as $q.in\_\ a$ = FALSE and $\texttt{IsMissing}(q)$ = TRUE in γ, by Corollary 3, we can conclude that $\texttt{MustJoin}(q)$ holds in γ, that is, q is enabled in γ, contradiction.

□

Lemma 34 *In any terminal configuration of* $\mathcal{MA}(f, g)$*, A is a* 1*-minimal* (f, g)*-alliance, and if* $f \geq g$*, then A is a minimal* (f, g)*-alliance.*

Proof. Let γ be a terminal configuration. We already know that in γ, A defines an (f, g)-alliance. Moreover, by Property 4, if A is 1-minimal and $f \geq g$, then A is a minimal (f, g)-alliance. Thus, we only need to show the 1-minimality of A.

Assume that A is not 1-minimal. Then there is a process $p \in A$ such that $A - \{p\}$ is an (f, g)-alliance. So:

1. $|A \cap \mathcal{N}(p)| \geq f(p)$,
2. $\forall q \in \mathcal{N}(p), q \in A \Rightarrow |A \cap \mathcal{N}(q) - \{p\}| \geq g(q)$, and
3. $\forall q \in \mathcal{N}(p), q \notin A \Rightarrow |A \cap \mathcal{N}(q) - \{p\}| \geq f(q)$.

By 1, in γ we have:

$$\texttt{NbA}(p) \geq f(p) \qquad (a)$$

By 2, in γ we have:

$$\begin{array}{l} \forall q \in \mathcal{N}(p), q.in\_\ a \Rightarrow \texttt{NbA}(q) - 1 \geq g(q) \\ \forall q \in \mathcal{N}(p), q.in\_\ a \Rightarrow \texttt{NbA}(q) > g(q) \\ \forall q \in \mathcal{N}(p), q.in\_\ a \Rightarrow q.nb\_\ a > g(q) \qquad \text{by Remark 6} \qquad (b) \end{array}$$

By 3, in γ we have:

$$\begin{array}{l} \forall q \in \mathcal{N}(p), \neg q.in\_\ a \Rightarrow \texttt{NbA}(q) - 1 \geq f(q) \\ \forall q \in \mathcal{N}(p), \neg q.in\_\ a \Rightarrow \texttt{NbA}(q) > f(q) \\ \forall q \in \mathcal{N}(p), \neg q.in\_\ a \Rightarrow q.nb\_\ a > f(q) \qquad \text{by Remark 6} \qquad (c) \end{array}$$

By (b) and (c), $\texttt{IsExtra}(p)$ holds in γ. So, by (a), $\texttt{NbA}(p) \geq f(p) \wedge \texttt{IsExtra}(p)$ holds in γ, that is, $\neg\texttt{IsBusy}(p)$ holds in γ. Now, Flag is disabled at p in γ , so $p.busy$ = FALSE in γ. As we assumed that $p.in\_\ a$ = TRUE in γ ($p \in A$), this contradicts Lemma 32. □

10.3.2.2 Termination

We now show that, if $f \geq g$, the unfair daemon cannot prevent $\mathcal{MA}(f, g)$ from terminating, starting from any configuration. The proof consists in showing that the number of steps to reach a terminal configuration, starting from any arbitrary configuration, is bounded, no matter the choices of daemon are.

Let J be the maximum number of times any process executes Action `Join` in any execution. Lemma 35, below, states that the number of steps to reach a terminal configuration of $\mathcal{MA}(f, g)$ depends on J, as well as on both global parameters of the network, its degree Δ, and its size n.

Lemma 35 *Starting from any configuration, $\mathcal{MA}(f, g)$ reaches a terminal configuration in $O(J\Delta^3 n)$ steps.*

Proof. Consider any process p in any execution e of $\mathcal{MA}(f, g)$. Let $J(p)$, $L(p)$, $C(p)$, $F(p)$, and $V(p)$ be the number of times p executes Actions `Join`, `Leave`, `Count`, `Flag` and `Vote` in e, respectively. By definition, $J(p) \leq J$.

After executing `Leave`, p should execute `Join` before executing `Leave` again. So:

$$L(p) \leq 1 + J(p) \leq 1 + J$$

In the following, we use the number of times p modifies the value of its variable $p.nb\_\,a$. This number is denoted by $\sharp nb\_\,a(p)$. $p.nb\_\,a$ is modified because either $p.nb\_\,a \neq \texttt{NbA}(p)$ in the initial configuration, or $p.nb\_\,a \neq \texttt{NbA}(p)$ becomes TRUE after a neighbor of p joins or leaves A. So:

$$\sharp nb\_\,a(p) \leq 1 + \textstyle\sum_{q \in \mathcal{N}(p)}(J(q) + L(q)) \leq 1 + \Delta(2J + 1)$$

By definition, p executes Action `Count` at most $\sharp nb\_\,a(p)$ times. So:

$$C(p) \leq \sharp nb\_\,a(p) \leq 1 + \Delta(2J + 1)$$

In the following, we use the number of times p modifies the value of its variable $p.busy$. This number is denoted by $\sharp busy(p)$. $p.busy$ is modified because either $p.busy \neq \texttt{IsBusy}(p)$ holds in the initial configuration, or $p.busy \neq \texttt{IsBusy}(p)$ becomes TRUE after a neighbor q of p joins or leaves A, or modifies its counter $q.nb\_\,a$. So:

$$\sharp busy(p) \leq 1 + \textstyle\sum_{q \in \mathcal{N}(p)}(J(q) + L(q) + \sharp nb\_\,a(q)) \leq 1 + (2 + 2J)\Delta + (1 + 2J)\Delta^2$$

By definition, p executes Action `Flag` at most $\sharp busy(p)$ times. So:

$$F(p) \leq \sharp busy(p) \leq 1 + (2 + 2J)\Delta + (1 + 2J)\Delta^2$$

Action `Vote` is enabled when p wants to change its pointer $p.choice$. That is, either (1) p does not want to authorize any neighbor to leave A (in this case, its pointer is reset to $\perp$), or (2) p has a new favorite candidate. In the latter case, p may be required to reset its pointer to $\perp$ first, because we impose a strict alternation in $p.choice$ between values of $\mathcal{N}(p)$ and $\perp$. Hence, p may require up to two executions of Action `Vote` to fix the value of $p.choice$.

As for other actions, `Vote` can be initially enabled. Moreover, either case (1) or (2) occurs for p every time either (i): the variables in_a of p or its neighbors are modified, or (ii): the variable $busy$ or nb_a of one or more of its neighbors is modified. Therefore

$$V(p) \leq 2(1 + \sum_{r \in \mathcal{N}(p) \cup \{p\}} (J(r) + L(r)) + \sum_{q \in \mathcal{N}(p)} (\sharp busy(q) + \sharp nb\_a(q)))$$
$$V(p) \leq 4 + 4J + \Delta(6 + 4J) + \Delta^2(6 + 8J) + \Delta^3(2 + 4J)$$

So, the maximum number of steps before $\mathcal{MA}(f, g)$ reaches a terminal configuration is:

$$\begin{aligned} & n(J(p) + L(p) + C(p) + F(p) + V(p)) \\ \leq\ & n[7 + 6J + \Delta(9 + 8J) + \Delta^2(7 + 10J) + \Delta^3(2 + 4J)] \\ =\ & O(J.\Delta^3.n) \end{aligned}$$

□

To complete the proof of convergence of $\mathcal{MA}(f, g)$, we now show, in Lemma 42, that J is bounded by 1 if $f \geq g$. This lemma uses six technical results, given in Lemmas 36 through 41.

Lemma 36 *Let p be a process. $\forall q, q' \in \mathcal{N}(p) \cup \{p\}$, if $q' \neq q$, then q and q' cannot leave A in the same step.*

Proof. By contradiction. Assume, that there are two processes $q, q' \in \mathcal{N}(p) \cup \{p\}$ such that $q' \neq q$, and both q and q' leave the alliance in some step $\gamma \mapsto \gamma'$. Consider the two following cases:

$q = p \vee q' = p$: Without loss of generality, assume that $q' = p$. From the guard of Action `Leave` at p, $p.choice = \perp$. Now, $p \in \mathcal{N}(q)$, so from the guard of Action `Leave` at q, $p.choice = q \neq \perp$, a contradiction.

$q \neq p \wedge q' \neq p$: By definition, $p \in \mathcal{N}(q)$ and $p \in \mathcal{N}(q')$. So, from the guard of Action `Leave` at q, we have $p.choice = q$; and from the guard of Action `Leave` at q', $p.choice = q'$, a contradiction.

□

Corollary 4 *If a process p leaves A in the step $\gamma \mapsto \gamma'$, then* `Fga`(p) *holds in γ'.*

Proof. Assume that process p leaves A in $\gamma \mapsto \gamma'$. From the guard of Action `Leave`, we have `NbA`$(p) \geq f(p)$. By Lemma 36, no neighbor of p leaves A in $\gamma \mapsto \gamma'$. So, $p.in\_a =$ FALSE and `NbA`$(p) \geq f(p)$ in γ', and we are done. □

Lemma 37 *If a process p executes* `Leave` *or* p.choice *is assigned the* UID *of some neighboring process in $\gamma \mapsto \gamma'$, then* $\texttt{NbAOk}(p)$ *holds in γ'.*

Proof. Let X be the value of $\texttt{NbA}(p)$ in γ.

If p executes `Leave` in $\gamma \mapsto \gamma'$, then from the guard of `Leave`, we know that $X \geq f(p)$. Moreover, as Action `Count` is disabled at p (otherwise, `Leave` is not executed because `Count` has higher priority), $p.nb\_\ a = X$ in γ. So, $p.in\_\ a =$ FALSE and $p.nb\_\ a = X \geq f(p)$ in γ', *i.e.*, $\texttt{NbAOk}(p)$ holds in γ'.

If p executes $p.choice \leftarrow q \in \mathcal{N}(p)$ in $\gamma \mapsto \gamma'$, then $\texttt{HasExtra}(p)$ holds in γ, p does not change the value of $p.in\_\ a$ in $\gamma \mapsto \gamma'$, and $p.nb\_\ a \leftarrow X$ in $\gamma \mapsto \gamma'$. Consequently, $\texttt{NbAOk}(p)$ holds in γ'. □

Lemma 38 *For every process p,* $\texttt{ChoiceOk}(p)$ *is closed.*

Proof. By contradiction. Assume that there is a process p such that $\texttt{ChoiceOk}(p)$ is not closed: There exists a step $\gamma_i \mapsto \gamma_{i+1}$ where $\texttt{ChoiceOk}(p)$ holds in γ_i, but not in γ_{i+1}. That is: $p.choice \neq \bot \wedge p.choice.in\_\ a \wedge \neg\texttt{HasExtra}(p)$ holds in γ_{i+1}.

Assume that the value of $p.in\_\ a$ changes between γ_i and γ_{i+1}. Then, p executes `Join` or `Leave` in $\gamma_i \mapsto \gamma_{i+1}$. In the former case, $p.choice = \bot$ in γ_{i+1}, and consequently, $\texttt{ChoiceOk}(p)$ still holds in γ_{i+1}, contradiction. In the latter case, from the guard of `Leave`, we can deduce that $p.choice = \bot$ in γ_i and, as Action `Leave` does not modify the variable *choice*, $p.choice = \bot$ still holds in γ_{i+1}, contradiction. So, the value of $p.in\_\ a$ does not change during $\gamma_i \mapsto \gamma_{i+1}$. Consider the following two cases:

A) $p.choice = \bot$ in γ_i: $p.choice \neq \bot$ in γ_{i+1}. So, p executes Action `Vote` in $\gamma_i \mapsto \gamma_{i+1}$. Consequently, the guard of Action `Vote` holds at p in γ_i. In particular, $\texttt{ChosenCand}(p) \neq \bot$ in γ_i, and so $\texttt{HasExtra}(p)$ also holds in γ_i. As the value of $p.in\_\ a$ does not change during $\gamma_i \mapsto \gamma_{i+1}$, a neighbor of p should leave A during $\gamma_i \mapsto \gamma_{i+1}$, so that $\texttt{HasExtra}(p)$ becomes FALSE. Since $p.choice = \bot$ in γ_i, no neighbor of p can execute Action `Leave` in $\gamma_i \mapsto \gamma_{i+1}$, contradiction.

B) $p.choice \neq \bot$ in γ_i: If p executes `Vote` in $\gamma_i \mapsto \gamma_{i+1}$, then $p.choice = \bot$ in γ_{i+1} and $\texttt{ChoiceOk}(p)$ still holds in γ_{i+1}, contradiction. So, the value of $p.choice$ is the same in γ_i and γ_{i+1}. Let q be this value. Recall that $q \in \mathcal{N}(p)$, and consider the following two subcases:

$\neg q.in\_\ a$ in γ_i: $q.in\_\ a$ holds in γ_{i+1}. So, q executes Action `Join` in $\gamma_i \mapsto \gamma_{i+1}$. Now, as $p.choice = q$ in γ_i, Action `Join` is disabled at q in γ_i, contradiction.

***q.in_ a* in γ_i:** Since `ChoiceOk`(p) holds in γ_i, we have `HasExtra`(p) = TRUE in γ_i. Now, `HasExtra`(p) is FALSE in γ_{i+1}. Moreover, we already know that the value of $p.in\_\ a$ does not change during $\gamma_i \mapsto \gamma_{i+1}$. So, by Lemma 36, exactly one neighbor of p executes Action `Leave` in $\gamma_i \mapsto \gamma_{i+1}$. As $p.choice = q$ in γ_i, the neighbor that leaves A in $\gamma_i \mapsto \gamma_{i+1}$ is necessarily q. So, $q.in\_\ a$ = FALSE in γ_{i+1}, and since $p.choice = q$ still holds in γ_{i+1}, we have $p.choice.in\_\ a$ = FALSE in γ_{i+1}. Consequently, `ChoiceOk`(p) still holds in γ_{i+1}, contradiction.

□

Lemma 39 *For every process p,* `ChoiceOk`(p) *holds forever after p executes any action.*

Proof. Let p be a process that executes any action in $\gamma \mapsto \gamma'$. By Lemma 38, we only need to show that `ChoiceOk`(p) is TRUE in either γ or γ'.

Consider the following three cases:

A) p executes `Join`: Then, $p.choice = \perp$ in γ', and consequently `ChoiceOk`(p) is TRUE in γ'.

B) p executes `Vote`: Then, $p.choice = \perp$ in either γ or γ', and `ChoiceOk`(p) is TRUE in γ or γ'.

C) p executes any other action: As in the previous cases, if $p.choice = \perp$ in γ, we conclude that `ChoiceOk`(p) is TRUE in γ. Suppose $p.choice \neq \perp$ in γ. Since `Join` and `Vote` have higher priority than any other action, we deduce that their respective guards are FALSE in γ. In particular, from the negation of the guard of Action `Vote`, we can deduce that $p.choice =$ `ChosenCand`$(p) \neq \perp$ in γ. So, `HasExtra`(p) holds in γ, and thus `ChoiceOk`(p) holds in γ. □

Lemma 40 *If $f \geq g$,* `ChoiceOk`$(p) \wedge$ `Fga`(p) *is closed for every process p.*

Proof. Let p be a process. Let $\gamma \mapsto \gamma'$ be any step such that `ChoiceOk`$(p) \wedge$ `Fga`(p) holds in γ. By Lemma 38, we have: *(*)* `ChoiceOk`(p) *holds in γ'.*

Hence, we only need to show that `Fga`(p) still holds in γ'. Let X be the value of `NbA`(p) in γ. Let Y be the value of `NbA`(p) in γ'. By Lemma 36, $Y \geq X - 1$. Consider the following two cases:

- *A) The value of p.*in_a *is the same in γ and γ'.*

  If $p.choice = \perp$ in γ, then no neighbor of p can leave A in $\gamma \mapsto \gamma'$. Consequently, $Y \geq X$, which also implies that `Fga`(p) still holds in γ'.

  Otherwise, $p.choice \neq \perp$ in γ. There are two cases.

  ***p.choice.in_ a* in γ:** By (*), $p.in\_\ a \Rightarrow X > g(p)$ and $\neg p.in\_\ a \Rightarrow X > f(p)$ in γ. So, as the value of $p.in\_\ a$ is the same in γ and γ', and $Y \geq X - 1$, we have $p.in\_\ a \Rightarrow Y \geq g(p)$ and $\neg p.in\_\ a \Rightarrow Y \geq f(p)$ in γ', which implies that `Fga`(p) still holds in γ'.

$\neg p.choice.in\_\ a$ in γ: There is no neighbor q of p such that $q.in\_\ a$ and $p.choice = q$ in γ. So, no neighbor of p leaves A in $\gamma \mapsto \gamma'$. Consequently, $Y \geq X$ and, as the value of $p.in\_\ a$ is the same in γ and γ', $\texttt{Fga}(p)$ still holds in γ'.

- *B) p changes the value of* p.in_a *in* $\gamma \mapsto \gamma'$. Consider the following two cases:

  p executes `Leave` in $\gamma \mapsto \gamma'$: First, $p.in\_\ a = \text{FALSE}$ in γ'. So, $\texttt{Fga}(p)$ holds in γ' only if $Y \geq f(p)$. Then, from the guard of Action `Leave`, we have (1) $X \geq f(p)$ and (2) $p.choice = \bot$ in γ. By (2), no neighbor of p leaves A in $\gamma \mapsto \gamma'$. So, $Y \geq X \geq f(p)$, which implies that $\texttt{Fga}(p)$ still holds in γ'.

  p executes `Join` in $\gamma \mapsto \gamma'$: First, $p.in\_\ a = \text{TRUE}$ in γ'. So, $\texttt{Fga}(p)$ holds in γ' only if $Y \geq g(p)$. (Recall that $f(p) \geq g(p)$.) Consider the following two cases:

  $X > Y$: Then $Y = X - 1$. Let q be the neighbor of p that leaves A in $\gamma \mapsto \gamma'$. $q.in\_\ a = \text{TRUE} \wedge p.choice = q$ in γ. So, by (*), $p.in\_\ a = \text{FALSE}$ in γ implies that $X > f(p)$. So, $Y \geq f(p) \geq g(p)$, which implies that $\texttt{Fga}(p)$ still holds in γ'.

  $X \leq Y$: Then, $Y \geq X \geq f(p) \geq g(p)$, which implies that $\texttt{Fga}(p)$ still holds in γ'.

□

Lemma 41 *Assuming $f \geq g$, we have: for every process p,* $\texttt{ChoiceOk}(p) \wedge \texttt{Fga}(p) \wedge \texttt{NbAOk}(p)$ *is closed.*

Proof. Let p be a process. Let $\gamma \mapsto \gamma'$ be any step such that $\texttt{ChoiceOk}(p) \wedge \texttt{Fga}(p) \wedge \texttt{NbAOk}(p)$ holds in γ. By Lemma 40, $\texttt{ChoiceOk}(p) \wedge \texttt{Fga}(p)$ is TRUE in γ'. So, we only need to show that $\texttt{NbAOk}(p)$ still holds in γ'.

Assume the contrary. Let X be the value of $\texttt{NbA}(p)$ in γ and consider the following two cases:

- *p does not change the value of* p.in_a *in* $\gamma \mapsto \gamma'$. Assume that $p.in\_\ a$ is TRUE in γ. Then, p must modify $p.nb\_\ a$ in $\gamma \mapsto \gamma'$ to violate $\texttt{NbAOk}(p)$ in γ'. From the algorithm, p executes $p.nb\_\ a \leftarrow X$ in $\gamma \mapsto \gamma'$. Then, $X \geq g(p)$ since $\texttt{Fga}(p)$ in γ. Thus, $p.in\_\ a = \text{TRUE}$ and $p.nb\_\ a \geq g(p)$ in γ', *i.e.*, $\texttt{NbAOk}(p)$ still holds in γ', contradiction.

  Assume that $p.in\_\ a$ is FALSE in γ. By similar reasoning, we obtain a contradiction in this case as well.

- *p changes the value of* p.in_a *in* $\gamma \mapsto \gamma'$. There are two cases:

  p leaves A in $\gamma \mapsto \gamma'$: Then, $\texttt{NbAOk}(p)$ still holds in γ' by Lemma 37, contradiction.

***p* joins** A **in** $\gamma \mapsto \gamma'$**:** Then, $X \geq f(p)$ because $p.in\_\ a = \text{FALSE}$ and $\texttt{Fga}(p)$ holds in γ. Then, $p.nb\_\ a \leftarrow X$ in $\gamma \mapsto \gamma'$. So, $p.in\_\ a = \text{TRUE}$ and $p.nb\_\ a \geq f(p) \geq g(p)$ in γ', *i.e.*, $\texttt{NbAOk}(p)$ still holds in γ', contradiction. □

Lemma 42 *If $f \geq g$, then in any execution of $\mathcal{MA}(f, g)$, $J \leq 1$, that is, every process joins the (f, g)-alliance at most once.*

(Figure 10.5 illustrates the following proof.)

Proof. By contradiction. Assume that some process p executes Action `Join` at least two times. Note that p must execute Action `Leave` between two executions of Action `Join`. Thus, there exist $0 \leq i < j < k$ such that p joins A in $\gamma_i \mapsto \gamma_{i+1}$, leaves A in $\gamma_j \mapsto \gamma_{j+1}$, and joins it again in $\gamma_k \mapsto \gamma_{k+1}$.

From the guard of Action `Join`, $q.choice \neq p$ in γ_i for all $q \in \mathcal{N}(p)$. From the guard of Action `Leave`, $q.choice = p$ in γ_j for all $q \in \mathcal{N}(p)$. Thus:

(1) Every neighbor q of p executes $q.choice \leftarrow p$ using Action `Vote` before γ_j.

Let q be any neighbor of p. Let $\gamma_l \mapsto \gamma_{l+1}$ be a step at which q executes $q.choice \leftarrow p$, using Action `Vote`, for $i < l < j$. Such a step exists by (1). By Lemma 39, $\texttt{ChoiceOk}(q)$ is TRUE in γ_{l+1}. Moreover, by (1) and the code of Action `Vote`, we can deduce that (a) $q.choice = \perp$ and (b) $p.in\_\ a = \text{TRUE}$ in γ_l. By (a), $p.in\_\ a$ is still TRUE in γ_{l+1}. Now, $q.choice = p$ in γ_{l+1}. So, $\texttt{ChoiceOk}(q)$ in γ_{l+1} implies that $\texttt{HasExtra}(q)$ holds in γ_{l+1}, which in turns implies that $\texttt{Fga}(q)$ holds in γ_{l+1}. Finally, $\texttt{NbAOk}(q)$ in γ_{l+1} by Lemma 37. So, by Lemma 41, $\texttt{ChoiceOk}(q) \wedge \texttt{Fga}(q) \wedge \texttt{NbAOk}(q)$ is true forever from γ_{l+1}. Hence:

(2) Every neighbor q of p satisfies $\texttt{ChoiceOk}(q) \wedge \texttt{Fga}(q) \wedge \texttt{NbAOk}(q)$ forever from γ_j.

As p leaves A in $\gamma_j \mapsto \gamma_{j+1}$, by Corollary 4 and Lemmas 39 and 40, we have:

(3) $\texttt{ChoiceOk}(p) \wedge \texttt{Fga}(p)$ holds forever from γ_{j+1}.

As p joins A in $\gamma_k \mapsto \gamma_{k+1}$, ($a$) $\neg p.in\_\ a \wedge \texttt{NbA}(p) < f(p)$ or (b) $\texttt{IsMissing}(p)$ holds in γ_k. Now, (a) contradicts (3) and (b) contradicts (2). □

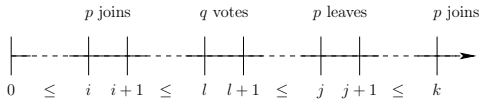


Figure 10.5 – Execution of $\mathcal{MA}(f, g)$ assuming p executes Action `Join` at least two times.

From Lemmas 35 and 42, we deduce the following corollary:

Corollary 5 *Starting from any configuration, if $f \geq g$, $\mathcal{MA}(f, g)$ reaches a terminal configuration in $O(n \times \Delta^3)$ steps.*

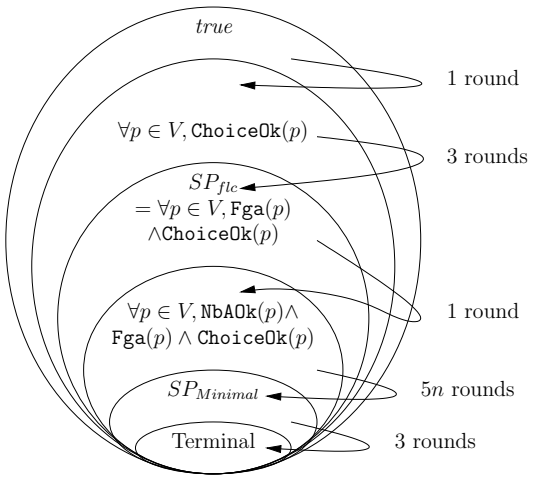


Figure 10.6 – Safe convergence of Algorithm $\mathcal{MA}(f, g)$.

By Lemma 34 and Corollary 5, we have:

Theorem 24 *If $f \geq g$, $\mathcal{MA}(f, g)$ is silent and self-stabilizing w.r.t. $Spec_{Minimal}$, and its stabilization time is $O(\Delta^3 n)$ steps.*

10.3.3 Safe Convergence and Complexity Analysis in Rounds

We define a *feasible legitimate configuration* to be any configuration γ that satisfies

$$Spec_{flc} \stackrel{\texttt{def}}{=} \forall p \in V,\ \texttt{ChoiceOk}(p) \wedge \texttt{Fga}(p)$$

In any feasible legitimate configuration, A is an (f, g)-alliance, by Remark 5. Then, from Lemma 40, we already know that the set of *feasible legitimate configurations* is closed if $f \geq g$:

Corollary 6 *If $f \geq g$, then $Spec_{flc}$ is closed.*

To establish safe convergence of $\mathcal{MA}(f, g)$, we show that it gradually converges to more and more specific closed predicates, until reaching a terminal configuration. The gradual convergence to those specific closed predicates is shown in Figure 10.6.

Lemma 43 *For every process p, after at most one round,* $\texttt{ChoiceOk}(p)$ *is* TRUE *forever.*

Proof. To show this lemma, it is sufficient to show that $\texttt{ChoiceOk}(p)$ becomes TRUE during the first round, by Lemma 38. If p is continuously enabled from the initial configuration, then p executes at least one action during the first round and by Lemma 39, we are done.

Otherwise, the first round contains a configuration γ in which every action is disabled at p. In particular, from the negation of the guard of Action `Vote`, we have $p.choice = \texttt{ChosenCand}(p)$ in γ. Two cases are then possible in γ:

***p.choice* $= \perp$:** In this case, by definition, $\texttt{ChoiceOk}(p)$ holds in γ.

***p.choice* $\neq \perp$:** Then, as $p.choice = \texttt{ChosenCand}(p)$, we have $p.choice = \texttt{MinCand}(p)$ in γ. Thus, $\texttt{HasExtra}(p)$ holds in γ, which implies that $\texttt{ChoiceOk}(p)$ holds in γ. □

Lemma 44 *Assume $f \geq g$. Let $\gamma_0 \dots \gamma_i \dots$ be an execution of $\mathcal{MA}(f, g)$. $\forall i \geq 0$, if $\texttt{ChoiceOk}(p)$ for all $p \in V$ in γ_i, then $\exists j \geq i$ such that γ_j is within at most three rounds from γ_i and $\forall p \in V, \texttt{ChoiceOk}(p) \wedge \texttt{Fga}(p)$ holds in γ_j.*

Proof. Let γ_{t_0} be a configuration where $\forall p \in V, \texttt{ChoiceOk}(p)$. Consider any execution (starting in γ_{t_0}) $e = \gamma_{t_0} \dots \gamma_{t_1} \dots \gamma_{t_2} \dots \gamma_{t_3} \dots$, where $\gamma_{t_1}, \gamma_{t_2}$, and γ_{t_3} are the last configurations of the first, second, and third rounds of e, respectively. By Lemma 38, it is sufficient to show that there is some $t \in [t_0..t_3]$ such that $\forall p \in V, \texttt{Fga}(p)$ in γ_t. Suppose no such a configuration exists. By Lemmas 38 and 40, this means that there exists a process v such that:

(1) $\forall t \in [t_0..t_3]$, $\neg\texttt{Fga}(v)$ in γ_t.

We now derive a contradiction using the following six claims.

(2) $\forall t \in [t_1..t_3]$, $v.choice = \perp$ in γ_t.

Proof of Claim 2: First, by (1), $\forall t \in [t_0..t_3]$, $\neg\texttt{HasExtra}(v)$ in γ_t. So, from the definition $\texttt{ChosenCand}(v)$, we can deduce that $\forall t \in [t_0..t_3]$, if $v.choice = \perp$ in γ_t, then $\forall t' \in [t..t_3]$, $v.choice = \perp$ in $\gamma_{t'}$. Hence, to show the claim, it is sufficient to show that $\exists t \in [t_0..t_1]$ such that $v.choice = \perp$ in γ_t. Suppose the contrary. Then, $\forall t \in [t_0..t_1]$, $v.choice \neq \perp \wedge \neg\texttt{HasExtra}(v)$ in γ_t, that is, the guard of `Vote` is TRUE at v in γ_t. So, v executes (at least) one of the two first actions in the first round to set $v.choice$ to $\perp$, and we are done.

(3) $\forall t \in [t_1..t_3]$, $\neg v.in\_\ a \Rightarrow (\forall q \in \mathcal{N}(v), q.choice \neq v)$ in γ_t.

Proof of Claim 3: Let $\gamma_t \mapsto \gamma_{t+1}$ such that $t \in [t_0..t_3 - 1]$. Assume that $\neg v.in\_\ a \Rightarrow (\forall q \in \mathcal{N}(v), q.choice \neq v)$ holds in γ_t.

If $v.in\_\ a =$ TRUE in γ_t, then $v.in\_\ a =$ TRUE in γ_{t+1} by (1) and Corollary 4, in particular, this implies that $\neg v.in\_\ a \Rightarrow (\forall q \in \mathcal{N}(v), q.choice \neq v)$ still holds in γ_{t+1}. Otherwise, $\neg v.in\_\ a \wedge (\forall q \in \mathcal{N}(v), q.choice \neq v)$ holds in γ_t and, from the definition of $\texttt{ChosenCand}(q)$, no neighbor of v can execute `Vote` to designate v with its pointer during $\gamma_t \mapsto \gamma_{t+1}$. Hence, $\neg v.in\_\ a \Rightarrow (\forall q \in \mathcal{N}(v), q.choice \neq v)$ still holds in γ_{t+1}.

Consequently, $\forall t \in [t_0..t_3]$, if $\neg v.in\_\ a \Rightarrow (\forall q \in \mathcal{N}(v), q.choice \neq v)$ holds in γ_t, then $\forall t' \in [t..t_3]$, $\neg v.in\_\ a \Rightarrow (\forall q \in \mathcal{N}(v), q.choice \neq v)$ still holds in $\gamma_{t'}$. Hence, to show this claim, it is sufficient to show that $\exists t \in [t_0..t_1]$ such that $\neg v.in\_\ a \Rightarrow (\forall q \in \mathcal{N}(v), q.choice \neq v)$ in γ_t. Assume the contrary: $\forall t \in [t_0..t_1]$, $\neg v.in\_\ a \wedge (\exists q \in \mathcal{N}(v), q.choice = v)$ holds in γ_t. Then, $\forall q \in \mathcal{N}(v)$, if $q.choice \neq v$ in γ_t with $t \in [t_0..t_1]$, then $\forall t' \in [t..t_1]$, $q.choice \neq v$ in $\gamma_{t'}$. So, v has a neighbor q

such that $\forall t \in [t_0..t_1]$, $q.choice = v$ in γ_t. Now, in this case, $\forall t \in [t_0..t_1]$, the guard of `Vote` is TRUE at q in γ_t. So, q executes (at least) one of the two first actions in the first round to set $q.choice$ to $\perp$, contradiction.

(4) $\forall t \in [t_2..t_3]$, $v.nb\_\ a \leq \texttt{NbA}(v)$ in γ_t.

Proof of Claim 4: First, by (2), no neighbor of v can leave the alliance during the second and third rounds, that is, $\texttt{NbA}(p)$ is monotonically nondecreasing during $[t_1..t_3]$. So, $\forall t \in [t_1..t_3]$, if $v.nb\_\ a \leq \texttt{NbA}(v)$ in γ_t, then $\forall t' \in [t..t_3]$, $v.nb\_\ a \leq \texttt{NbA}(v)$ in $\gamma_{t'}$. Hence, to show this claim, it is sufficient to show that $\exists t \in [t_1..t_2]$ such that $v.nb\_\ a \leq \texttt{NbA}(v)$ in γ_t. Assume the contrary, namely that $v.nb\_\ a > \texttt{NbA}(v)$ in γ_t, $\forall t \in [t_1..t_2]$. Then, $\forall t \in [t_1..t_2]$, the guard of `Count` is TRUE at v. Consequently, v executes one of the three first actions, in particular $v.nb\_\ a \leftarrow \texttt{NbA}(v)$, during the second round, and, as $\texttt{NbA}(p)$ is monotonically nondecreasing during $[t_1..t_3]$, we obtain a contradiction.

(5) $\forall t \in [t_2..t_3]$, $v.in\_\ a$ in γ_t.

Proof of Claim 5: First, $\forall t \in [t_0..t_3]$, if $v.in\_\ a =$ TRUE in γ_t, then $\forall t' \in [t..t_3]$, $v.in\_\ a =$ TRUE in $\gamma_{t'}$ by (1) and Corollary 4. Hence, to show this claim, it is sufficient to show that $\exists t \in [t_0..t_2]$ such that $v.in\_\ a =$ TRUE in γ_t. Assume the contrary: $\forall t \in [t_0..t_2]$, $v.in\_\ a =$ FALSE in γ_t. Then, by (1) $\forall t \in [t_0..t_2]$, $\texttt{NbA}(v) < f(v)$ in γ_t. Now, by (3), $\forall t \in [t_1..t_3]$, $\forall q \in \mathcal{N}(v), q.choice \neq v$ in γ_t. So, the guard of the highest priority action of v, `Join`, is true in particular in every configuration γ_t where $t \in [t_1..t_2]$. So, v joins the alliance in the second round, contradiction.

(6) $\forall t \in [t_2..t_3]$, $\forall q \in \mathcal{N}(v), \neg q.in\_\ a \Rightarrow (\forall r \in \mathcal{N}(q), r.choice \neq q)$ in γ_t.

Proof of Claim 6: Let q be a neighbor of v. Let $\gamma_t \mapsto \gamma_{t+1}$ such that $t \in [t_1..t_3 - 1]$. Assume that $\neg q.in\_\ a \Rightarrow (\forall r \in \mathcal{N}(q), r.choice \neq q)$ holds in γ_t.

If $q.in\_\ a =$ TRUE in γ_t, then by (2), the guard of `Leave` is disabled at q, so $q.in\_\ a =$ TRUE in γ_{t+1}, and consequently, $\neg q.in\_\ a \Rightarrow (\forall r \in \mathcal{N}(q), r.choice \neq q)$ still holds in γ_{t+1}. Otherwise, $\neg q.in\_\ a \wedge (\forall r \in \mathcal{N}(q), r.choice \neq q)$ holds in γ_t and, from the definition of $\texttt{ChosenCand}(r)$, no neighbor r of q can execute `Vote` to designate q with its pointer during $\gamma_t \mapsto \gamma_{t+1}$. Hence, $\neg q.in\_\ a \Rightarrow (\forall r \in \mathcal{N}(q), r.choice \neq q)$ still holds in γ_{t+1}.

Consequently, $\forall t \in [t_1..t_3]$, $\forall q \in \mathcal{N}(v)$, if $\neg q.in\_\ a \Rightarrow (\forall r \in \mathcal{N}(q), r.choice \neq q)$ holds in γ_t, then $\forall t' \in [t..t_3]$, $\neg q.in\_\ a \Rightarrow (\forall r \in \mathcal{N}(q), r.choice \neq q)$ holds in $\gamma_{t'}$. Hence, to show this claim, it is sufficient to show that $\forall q \in \mathcal{N}(v)$, $\exists t \in [t_1..t_2]$ such that $\neg q.in\_\ a \Rightarrow (\forall r \in \mathcal{N}(q), r.choice \neq q)$ in γ_t. Assume the contrary: let q be a neighbor of v such that $\forall t \in [t_1..t_2]$, $\neg q.in\_\ a \wedge (\exists r \in \mathcal{N}(q), r.choice = q)$ holds in γ_t. First, $\forall r \in \mathcal{N}(q)$, if $r.choice \neq q$ in γ_t with $t \in [t_1..t_2]$, then $\forall t' \in [t..t_2]$, $r.choice \neq q$. So, there is a neighbor r of q that $\forall t \in [t_1..t_2]$, $r.choice = q$. Then, from the definition of $\texttt{ChosenCand}(r)$, $\forall t \in [t_1..t_2]$, the guard of `Vote` is TRUE at r in γ_t. So, r executes (at least) one of the two first actions in the second round to set $r.choice$ to $\perp$, a contradiction.

(7) $\forall q \in \mathcal{N}(v), q.in\_\ a$ in γ_{t_3}.

Proof of Claim 7: Let q be a neighbor of v. By (2), $\forall t \in [t_2..t_3]$, $\texttt{CanLeave}(q) = \text{FALSE}$. So, $\forall t \in [t_2..t_3]$, if $q.in\_\ a$ in γ_t, then $\forall t' \in [t..t_3]$, $q.in\_\ a$ in $\gamma_{t'}$. Hence, to show this claim, it is sufficient to show that $\exists t \in [t_2..t_3]$ such that $q.in\_\ a$ in γ_t. Assume the contrary: $\forall t \in [t_2..t_3]$, $\neg q.in\_\ a$. By (1) and (4), $\forall t \in [t_2..t_3]$, $\texttt{IsMissing}(q)$ holds in γ_t. Then, using (6), we deduce that the guard of the highest priority action of q, `Join`, is true in every configuration γ_t with $t \in [t_2..t_3]$. So, q joins the alliance in the third round, contradiction.

By (5), (7), and the fact that $\delta(v) \geq g(v)$, $\texttt{Fga}(v)$ holds in γ_{t_3}, a contradiction. □

By Remark 5, Lemmas 40, 43, and 44, we have the following:

Corollary 7 *If $f \geq g$, $\mathcal{MA}(f, g)$ is self-stabilizing w.r.t. $Spec_{flc}$, and the first convergence time of $\mathcal{MA}(f, g)$ is at most four rounds.*

Lemma 45 *If $f \geq g$, then from any configuration where $\forall p \in V, \texttt{ChoiceOk}(p) \wedge \texttt{Fga}(p) \wedge \texttt{NbAOk}(p)$, Action `Join` is forever disabled at every process.*

Proof. Let γ by any configuration where $\forall p \in V, \texttt{ChoiceOk}(p) \wedge \texttt{Fga}(p) \wedge \texttt{NbAOk}(p)$. Then, $\texttt{Fga}(p)$ implies that $\neg p.in\_\ a \Rightarrow \texttt{NbA}(p) \geq f(p)$ in γ. Moreover, $(\forall q \in \mathcal{N}(p), \texttt{Fga}(q) \wedge \texttt{NbAOk}(q))$ implies $\neg\texttt{IsMissing}(p)$ in γ. So, Action `Join` is disabled at every process p in γ. By Lemma 41, we are done. □

Lemma 46 *Let γ be any configuration where $\forall p \in V, \texttt{ChoiceOk}(p) \wedge \texttt{Fga}(p)$. If $f \geq g$, a configuration where $\forall p \in V, \texttt{ChoiceOk}(p) \wedge \texttt{Fga}(p) \wedge \texttt{NbAOk}(p)$ is forever* TRUE *is reached in at most one round from γ.*

Proof. By Lemmas 40 and 41, it is sufficient to show that $\forall p \in V$, there is a configuration in the first round starting from γ where $\texttt{NbAOk}(p)$ holds. Let p be a process. Consider the following two cases:

- *The value of* $p.\text{in\_a}$ *changes during the first round from γ.* If p leaves A, then by Lemma 37, we are done. Otherwise, p executes `Join` in some step $\gamma' \mapsto \gamma''$ of the round. So, $\texttt{NbA}(p) \geq f(p)$ in γ' (Lemma 40) and consequently, $p.nb\_\ a \geq f(p)$ in γ''. As $f(p) \geq g(p)$ and $p.in\_\ a = \text{TRUE}$ in γ'', we are done.

- *The value of* $p.\text{in\_a}$ *does not change during the first round from γ.* Assume that $\texttt{NbAOk}(p) = \text{FALSE}$ in all the configurations of the first round from γ. Then, as $\texttt{Fga}(p)$ is always TRUE (Lemma 40), the guard of Action `Count` is always TRUE during this round, and consequently p executes at least one of its three first actions in the round, in particular, $p.nb\_\ a \leftarrow \texttt{NbA}(p)$. Again, as $\texttt{Fga}(p)$ is always TRUE during the round (Lemma 40), we obtain a contradiction, and thus we are done. □

Lemma 47 *If $f \geq g$, then from any configuration where $(\forall p \in V, \texttt{ChoiceOk}(p) \wedge \texttt{Fga}(p) \wedge \texttt{NbAOk}(p))$, and A is not a 1-minimal (f,g)-alliance, at least one process permanently leaves A every five rounds.*

Proof. By contradiction. Let γ_{t_0} be a configuration where $\forall p \in V, \texttt{ChoiceOk}(p) \wedge \texttt{Fga}(p) \wedge \texttt{NbAOk}(p)$. Consider any execution (starting in γ_{t_0}) $e = \gamma_{t_0} \ldots \gamma_{t_1} \ldots \gamma_{t_2} \ldots \gamma_{t_3} \ldots \gamma_{t_4} \ldots \gamma_{t_5} \ldots$, where γ_{t_1}, γ_{t_2}, γ_{t_3}, γ_{t_4}, γ_{t_5} respectively are the last configurations of the first, second, third, fourth, fifth round of e. By Lemma 45, it is sufficient to show that $\exists t \in [t_0..t_5 - 1]$ such that some process leaves the alliance during $\gamma_t \mapsto \gamma_{t+1}$. Assume that no such a configuration exists.

Let $S = \{p \in V, p.in\_\ a \wedge \texttt{NbA}(p) \geq f(p) \wedge (\forall q \in \mathcal{N}(p), \texttt{HasExtra}(q))\}$. As A is not a 1-minimal (f,g)-alliance during the five first rounds after γ_{t_0}, $S \neq \emptyset$. Moreover, as no process leaves (by hypothesis) or joins (by Lemma 45) the alliance during the five first rounds from γ_{t_0}, S is constant during these rounds. Let $p_{\min} = \min(S)$.

We derive a contradiction, using the following six claims:

(1) $\forall t \in [t_1..t_5], \forall p \in V, p.nb\_\ a = \texttt{NbA}(p)$ in γ_t.

Proof of Claim 1: First, by hypothesis, $\forall p \in V$, the value of $\texttt{NbA}(p)$ is constant during the five first rounds. So, to show the claim, it is sufficient to prove that $\forall p \in V$, $\exists t \in [t_0..t_1]$, $p.nb\_\ a = \texttt{NbA}(p)$ in γ_t. Assume the contrary: there is a process p such that $\forall t \in [t_0..t_1]$, $p.nb\_\ a \neq \texttt{NbA}(p)$ in γ_t. Then, $\forall t \in [t_0..t_1]$, the guard of `Count` is TRUE at p. As Action `Join` is disabled forever at p (by Lemma 45), p executes the second or third actions, in particular $p.nb\_\ a \leftarrow \texttt{NbA}(p)$, during the first round, and we obtain a contradiction.

(2) $\forall t \in [t_1..t_5]$, $\texttt{IsBusy}(p_{\min}) =$ FALSE in γ_t.

Proof of Claim 2: From (1) and the definition of $p_{\min}$.

(3) $\forall t \in [t_2..t_5]$, $p_{\min}.choice = \perp$ in γ_t.

Proof of Claim 3: By (2) and the definition of $p_{\min}$, $\forall t \in [t_1..t_5]$, $\texttt{IamCand}(p_{\min})$ is TRUE but $\texttt{MinCand}(p_{\min}) < p_{\min}$ is FALSE in γ_t. So, $\forall t \in [t_1..t_5]$, $\texttt{ChosenCand}(p_{\min}) = \perp$ in γ_t. Hence to show the claim, it is sufficient to prove that $\exists t \in [t_1..t_2]$, $p_{\min}.choice = \perp$ in γ_t. Assume the contrary: $\forall t \in [t_1..t_2]$, $p_{\min}.choice \neq \perp$ in γ_t and consequently the guard of Action `Vote` is TRUE in γ_t. Now, $\forall t \in [t_1..t_2]$, `Join` is disabled at $p_{\min}$ in γ_t by Lemma 45. So, $p_{\min}$ executes Action `Vote` during the second round, and we are done.

(4) $\forall t \in [t_2..t_5]$, $\neg p_{\min}.busy$ in γ_t.

Proof of Claim 4: By (2), if $\exists t \in [t_1..t_5]$ such that $\neg p_{\min}.busy$ in γ_t, then $\forall t' \in [t..t_5]$, $\neg p_{\min}.busy$ in $\gamma_{t'}$. Hence to show the claim, it is sufficient to prove that $\exists t \in [t_1..t_2]$ such that $\neg p_{\min}.busy$ in γ_t. Assume the contrary: $\forall t \in [t_1..t_2]$, $p_{\min}.busy =$ TRUE in γ_t. $\forall t \in [t_1..t_2]$, `Join` and `Count` are disabled at $p_{\min}$ in γ_t (Lemma 45 and (1)). By (2), $\forall t \in [t_1..t_2]$, the guard of Action `Flag` is TRUE at $p_{\min}$ in γ_t. Consequently, $p_{\min}$ executes `Vote` or `Flag` during the second round, and we are done.

(5) $\forall t \in [t_3..t_5]$, $\forall q \in \mathcal{N}(p_{\min}), q.choice \in \{\perp, p_{\min}\}$ in γ_t.

Proof of Claim 5: By (4) and the definition of $p_{\min}$, $\forall t \in [t_2..t_5]$, $\forall q \in \mathcal{N}(p_{\min})$, $\texttt{ChosenCand}(q) = p_{\min}$ in γ_t. Hence, to show the claim, it is sufficient to prove that $\forall q \in \mathcal{N}(p_{\min})$, $\exists t \in [t_2..t_3]$ such that $q.choice \in \{\perp, p_{\min}\}$ in γ_t. Assume the contrary: let q be a neighbor of $p_{\min}$, and assume that $\forall t \in [t_2..t_3]$, $q.choice \notin \{\perp, p_{\min}\}$ in γ_t. Then, the guard of Action `Vote` is TRUE at q in γ_t. Now, $\forall t \in [t_2..t_3]$, `Join` is disabled at q in γ_t, by Lemma 45. So, q executes Action `Vote` during the second round, and we are done.

(6) $\forall t \in [t_4..t_5]$, $\forall q \in \mathcal{N}(p_{\min}), q.choice = p_{\min}$ in γ_t.

Proof of Claim 6: By (4) and the definition of $p_{\min}$, $\forall t \in [t_3..t_5]$, $\forall q \in \mathcal{N}(p_{\min})$, $\texttt{ChosenCand}(q) = p_{\min}$ in γ_t. Hence to show the claim, it is sufficient to prove that $\forall q \in \mathcal{N}(p_{\min})$, $\exists t \in [t_3..t_4]$, $q.choice = p_{\min}$ in γ_t. Assume the contrary: Let q be a neighbor of $p_{\min}$. Assume that $\forall t \in [t_3..t_4]$, $q.choice \neq p_{\min}$ in γ_t. Then, $\forall t \in [t_3..t_4]$, $q.choice = \perp$ in γ_t by (5) and consequently the guard of Action `Vote` is TRUE at q in γ_t. Now, $\forall t \in [t_3..t_4]$, `Join` is disabled at q in γ_t, by Lemma 45. So, q executes Action `Vote` during the third round and we are done.

From γ_{t_0}, Action `Join` is disabled at $p_{\min}$ forever. By (3), (4), and the definition of $p_{\min}$, $\forall t \in [t_4..t_5]$ Action `Vote` is disabled at $p_{\min}$. By (1), $\forall t \in [t_4..t_5]$ Action `Count` is disabled at $p_{\min}$. By (2) and (4), $\forall t \in [t_4..t_5]$ Action `Flag` is disabled at $p_{\min}$. By (3), (6), and the definition of $p_{\min}$, $\forall t \in [t_4..t_5]$, `Leave` is enabled at $p_{\min}$. So, $p_{\min}$ leaves the alliance during the fifth round, contradiction. □

Theorem 25 *If $f \geq g$, $\mathcal{MA}(f, g)$ is silent and self-stabilizing w.r.t. $Spec_{1-Minimal}$ and its stabilization time is at most $5n + 8$ rounds.*

Proof. By Lemmas 43 through 47, starting from any configuration, the system reaches a configuration γ from which A is a 1-minimal (f, g)-alliance and Actions `Join` and `Leave` are disabled forever at every process, in $5n+5$ rounds. So, it remains to show that the system reaches a terminal configuration after at most three rounds from γ.

The following three claims establish the proof:

(1) After one round from γ, $\forall p \in V$, $p.nb\_a = \texttt{NbA}(p)$ forever.

Proof of Claim 1: From γ, for every process p, `Join` is disabled forever and $\texttt{NbA}(p)$ is constant. So, if necessary, p fixes the value of $p.nb\_a$ to $\texttt{NbA}(p)$ within the next round by `Vote` or `Count`.

(2) After two rounds from γ, $\forall p \in V$, $(p.in\_\ a \Rightarrow p.busy) \wedge p.busy = \texttt{IsBusy}(p)$ forever.

Proof of Claim 2: When the second round from γ begins, for every process p, values of $p.in\_\ a$ and $p.nb\_\ a$ are constant, moreover `Join` and `Count` are disabled forever at p (by hypothesis and claim (1)). So, if necessary, p fixes the value of $p.busy$ to $\texttt{IsBusy}(p)$ within the next round by `Vote` or `Flag`. Hence, after two rounds from γ, $\forall p \in V$, $p.busy = \texttt{IsBusy}(p)$ holds forever.

Finally, assume that there is a process p such that $p.in\_\ a \wedge \neg p.busy$ after two rounds from γ. Then, $p.in\_\ a \wedge \texttt{NbA}(p) \geq f(p) \wedge \texttt{IsExtra}(p)$. Now, by (1), this means that $p.in\_\ a \wedge \texttt{NbA}(p) \geq f(p) \wedge (\forall q \in \mathcal{N}(p), (\neg q.in\_\ a \Rightarrow \texttt{NbA}(q) > f(q)) \wedge (q.in\_\ a \Rightarrow \texttt{NbA}(q) > g(q)))$, which contradicts the fact that A is a 1-minimal (f, g)-alliance. Hence, after two rounds from γ, $\forall p \in V$, $(p.in\_\ a \Rightarrow p.busy)$ holds forever.

(3) After three rounds from γ, $\forall p \in V$, $p.choice = \perp$ forever.

Proof of Claim 3: When the third round from γ begins, for every process p, $\texttt{Cand}(p) = \emptyset$ forever by Claim (2), which implies that $\texttt{ChosenCand}(p) = \perp$ forever. Remember also that `Join` is disabled forever for every process. So, if necessary, p fixes the value of $p.choice$ to $\perp$ within the next round by `Vote`.

From the three previous claims, we can deduce that after at most three rounds from γ (that is, at most $5n + 8$ rounds from the initial configuration), the system reaches a terminal configuration where $Spec_{Minimal}$ holds, by Lemma 34. □

By Property 4, Corollary 7, and Theorem 25, we have:

Corollary 8 *If $f \geq g$, $\mathcal{MA}(f, g)$ is silent and safely converging self-stabilizing w.r.t. $(Spec_{flc}, Spec_{Minimal})$, its first convergence time is at most four rounds, its second convergence time is at most $5n + 4$ rounds, and its stabilization time is at most $5n + 8$ rounds.*

Part V

Ranking

Chapter 11

Ranking in Ordered Trees

Contents

In this chapter, we give a self-stabilizing algorithm for tree networks that solves the *ranking problem* for an ordered tree, where each process has an *input value*, in $O(n)$ rounds, and has space complexity $O(\delta(p) \log n)$ in each process p.

The only previous self-stabilizing algorithm for the ranking problem is given in [BDN95]. This algorithm works in rooted trees. Like ours, that algorithm is not silent. It assumes that each process has a unique identifier in the range $[1..n]$. The algorithm stabilizes in $O(nh)$ rounds using $O(\log n)$ space per process, where h is the height of the tree and n the number of processes in the network.

The ranking problem is related to the *sorting problem* where each process is given an input value and must hold a final value, such that the set of final values is the set of input values, sorted over the network. There are numerous self-stabilizing solutions for sorting in a tree, *e.g.*, [HP01, HM01, BDV05].

Our algorithm, $\mathcal{RANK}$, makes use of the guide pairs computed by algorithm $\mathcal{GUIDE}$ presented in Chapter 5. The input of algorithm $\mathcal{RANK}$ consists of a weight $p.weight$, of some ordered type, for each process p. $\mathcal{RANK}$ computes the *rank* of each process, *i.e.*, the process of smallest weight is given rank 1, the second smallest rank 2, and so forth.

$\mathcal{RANK}$ correctly computes the rank of every process within $O(n)$ rounds, but is not silent. The ranks do not change once the system stabilizes. However, the algorithm repeatedly computes those ranks. If the weights do not change, the repeated computation of $\mathcal{RANK}$ will be transparent to any application that uses the output of $\mathcal{RANK}$.

11.1 Algorithm $\mathcal{RANK}$

We are given an ordered tree T and a value $p.weight$ for each process p in T. For convenience, we assume, in the discussion, that the weights are integers. The *ranking problem* is to find the *rank* of each p. If $p_1, p_2, \ldots, p_n$ is the list of processes in T sorted by weight, then ρ is the rank of p_ρ.

Our algorithm $\mathcal{RANK}$ is a hierarchical collateral composition of two algorithms: $\mathcal{RANK} = \mathcal{CRK} \circ \mathcal{GUIDE}$. $\mathcal{RANK}$ computes the rank of each process p in T, and sets the variable $p.rank$ to that value. We assume a weakly fair daemon for both algorithms. We also assume that, for every process p in T, for every neighbor q of p in T, p can determine whether p is the parent of q in T, *i.e.*, $par(q)$. To this purpose, we use the same implementation than in Chapter 5, that is, the variable $q.parent$ for every process q such that $q.parent = par(q)$ and the macro $\texttt{Children}(q) = \{u \in \mathcal{N}(q) : u.parent = q\}$.

11.2 Overview of Algorithm $\mathcal{CRK}$

11.2.1 Flow of Packages

The key part of the algorithm $\mathcal{CRK}$ is the *flow of packages* in T. Each *package* is an ordered pair $x = (x.value, x.guide)$, where $x.value$ is its *value* and $x.guide$ is its *guide pair*. Moreover, for any two packages x and y, we say $x > y$ if $x.value > y.value$.

Each package has a *home process* (the process from which the package is originally issued), although its *host* (location) can be at any process in the chain between its home and the root. Each process can host up to two packages: one *up-package*, that is moving toward the root, and one *down-package*, that is moving back to its home. The guide pair of a package is the same as the guide pair of its home process, and its value is the weight of its home process if it is an up-package, and the rank that $\mathcal{CRK}$ will assign to its home process if it is a down-package.

Each process p initiates its flow of packages by creating an up-package whose value is $p.weight$. This up-package then moves to the root by forward copying. The flow of packages is organized so that packages with smaller weights reach the root before packages with larger weights, in a manner similar to the standard technique for maintaining *minimum heap-order* in a tree [Wil64].

After the root copies an up-package from a child, it creates a down-package with the same home process as the up-package, but whose value is a number (a rank) in the range $1..n$. The root maintains a counter so that the first down-package it creates has value 1, the second value 2, and so forth. Each down-package then moves back to its home process by forward copying. When its home process copies a down-package, it assigns, or re-assigns, its rank to be the value of that package.

Since the root copies up-packages in weight order, it creates down-packages in that same order. The ρ^{th} down-package created by the root will carry rank ρ and will use the same guide pair as the ρ^{th} up-package copied by the root. Its home process will then be the process whose weight is the ρ^{th} smallest in T.

$\mathcal{CRK}$ is not silent, but rather, endlessly repeats its computation. When the root detects that it has created all down-packages, it initiates a broadcast wave which resets the variables of $\mathcal{CRK}$ (except the rank and weight variables) and the computation of ranks starts over.

11.2.2 Redundant Packages

In our model of computation, if a variable of a process p is copied by a neighbor q, it also remains at p. In the algorithm $\mathcal{CRK}$, each process p can be home to at most one package, but we cannot avoid the existence of multiple copies of that package (up and/or down). We handle that problem by defining a package variable currently hosted by a process as being either *active* or *redundant*. A redundant package can be overwritten, but not an active package.

If x is an up-package currently hosted by some process q which is not the root, then x is redundant if x has already been copied by $q.parent$. If x is an up-package currently hosted by the root, then x is redundant if the root has already created a down-package with the same guide pair as x. Any other up-package is active.

If x is a down-package hosted by some process q which is not its home process, then x is redundant if it has been copied by some child of q. (The child that copies x must be the process whose subtree contains the home process of x.) If x is a down-package hosted by its home process p, then x is redundant if $p.rank$ is equal to the value of x, indicating that p has already copied its rank from x, or that $p.rank$ was correct before x arrived. Any other down-package is active.

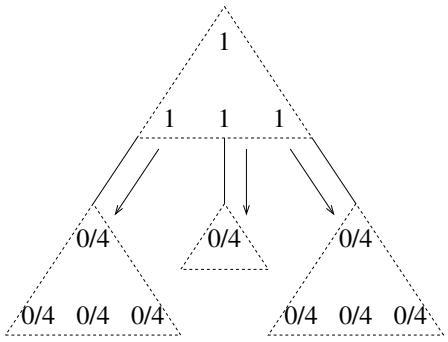
11.2.3 Status Waves

As it is typical for distributed algorithms which are self-stabilizing, but not silent, $\mathcal{CRK}$ endlessly repeats the calculation of the ranks of the processes in T. We call one (complete) pass through this cycle of computations an *epoch*. At the end of each epoch, the variables of $\mathcal{CRK}$ at all processes, other than the variables for weight and rank, are reset for the next epoch. If an epoch has a "clean start", it will calculate the correct rank for each process. Subsequent epochs will simply recalculate the same value, and $p.rank$ will never change again. Thus, the ranks will eventually appear constant to the application.

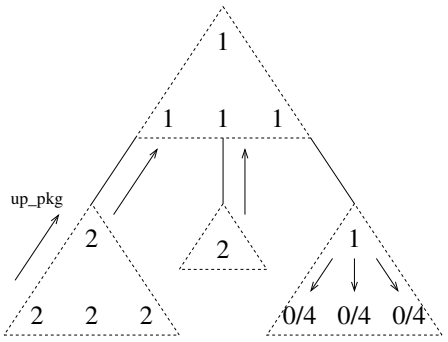
On the other hand, in case of an arbitrary initial configuration, it is possible for incorrect values of rank to be calculated during the first epoch, but eventually a configuration will be reached where the next epoch will get a clean start.

This system is controlled by the *status* variables of the processes. Status management is illustrated in Figure 11.1. At the beginning of an epoch, a broadcast wave starting from the root changes the status of every process from either 0 or 4 to 1, (Figure 11.1a), and all variables of $\mathcal{CRK}$ except rank and weight are set to their initial values. When this wave reaches the leaves of T, a convergecast wave changes the status of all processes to 2 (Figure 11.1b). All computation of the ranking algorithm, as discussed above, takes place while processes have status 2. Once a process p

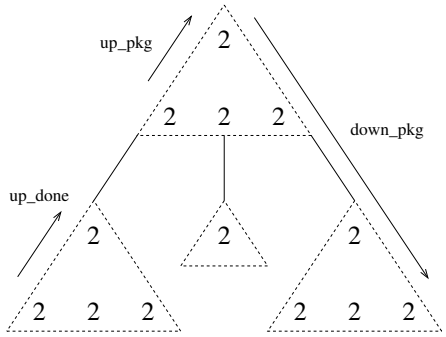
detects that all processes in its subtree have created their own up-package and the subtree no longer contains any up-packages, it sets its Boolean variable *p.up_ done* to TRUE (Figure 11.1c). After r has created the last down-package, it also satisfies *r.up_ done* = TRUE, and consequently initiates a broadcast wave where the status of all processes change to 3 (Figure 11.1d). A process propagates the status 3 once its last down-package becomes redundant. The return convergecast wave then changes the status of all processes to 4 (Figure 11.1e), and when this wave reaches the root, the new epoch begins (Figure 11.1f).



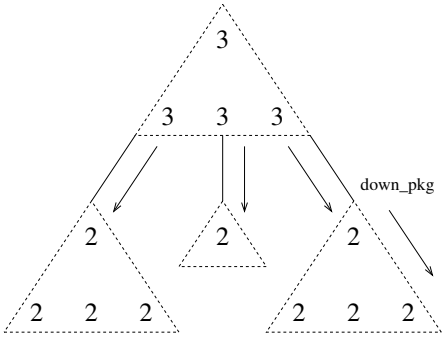
(a) Initial broadcast wave of status 1 resets variables.



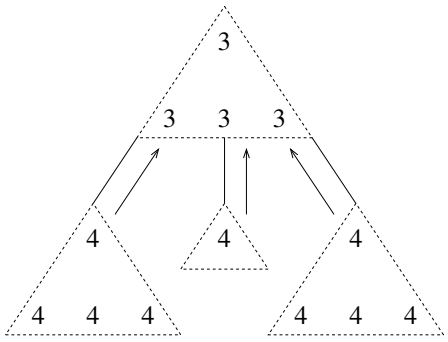
(b) Convergecast wave to status 2 allows for up-package propagation.



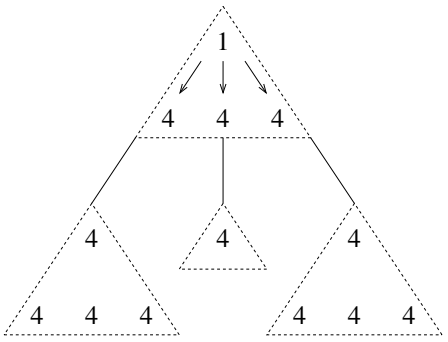
(c) Computation continues until *r.up_ done* = TRUE.



(d) The last down-packages are pushed by broadcast wave of status 3.



(e) The final convergecast wave of status 4 acknowledges receipt of all down-packages.



(f) Once r has status 4, it starts a new epoch with a status 1 broadcast wave.

Figure 11.1 – Status waves for a complete cycle of computations.

Status zero is used for error correction. If any process detects that the current epoch is erroneous, it changes its status to 0. Status 0 spreads down the tree, as well as up the tree unless it meets a process whose status is 1. If $r.status$ becomes 0 (and all its children have status 0 or 4), then r initiates a status 1 broadcast wave starting a new epoch. However, we must prevent an endless cycle of 0 and 1 waves going up and down the tree respectively. We solve this problem by adding a special rule for the non-root processes. If $p.status = 0$ and $p.parent.status = 1$, the status 0 wave cannot move up; instead, the status 0 wave moves only down, followed by the status 1 wave. This is illustrated in Figures 11.2 and 11.3.

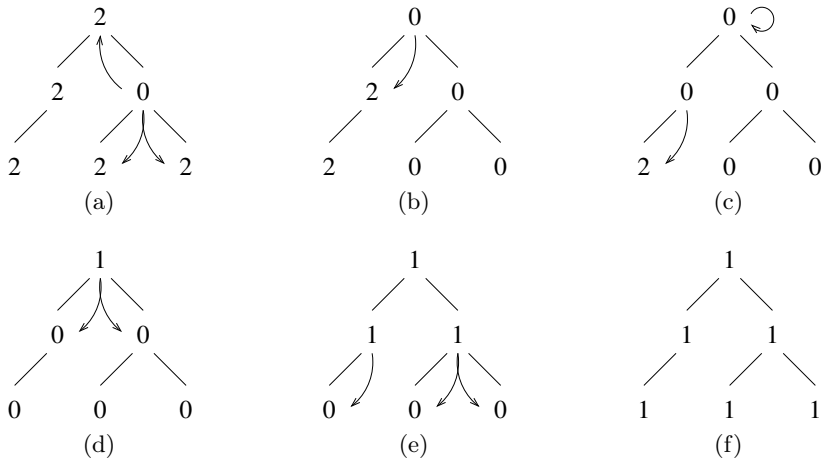


Figure 11.2 – Error correction when root process gets status 0.

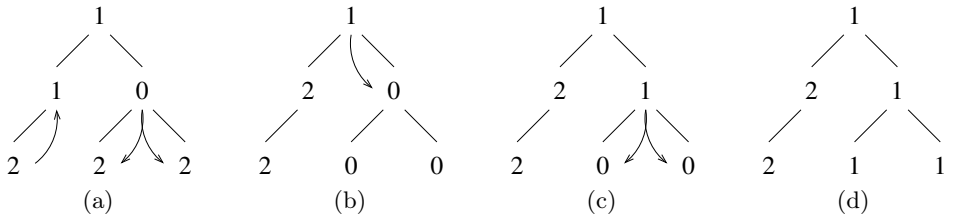


Figure 11.3 – Error correction when r already has status 1.

11.3 Formal Definition of Algorithm $\mathcal{CRK}$

11.3.1 Variables of $\mathcal{CRK}$

Let p be any process. Recall that $T(p)$ is the subtree of T rooted at p. $p.parent$, $p.guide$, and $p.weight$ are inputs of $\mathcal{CRK}$. Then, the output of $\mathcal{CRK}$ is $p.rank$, an integer. To compute this output, p maintains the following additional variables:

1. $p.up\_pkg$ and $p.down\_pkg$ are respectively of package type (that is, a guide pair and an integer) or $\perp$ (undefined).

   If $p.up\_pkg$ (resp. $p.down\_pkg$) is defined, then its home process is some $q \in T(p)$.

2. $p.started$, Boolean.

   This variable indicates that p has already generated its up-package during this epoch. ($p.up\_pkg$ may or may not still contain that up-package.)

3. $p.up\_done$, Boolean.

   This variable is TRUE if all processes in $T(p)$ have created their up-packages, and all such active up-packages have moved above p. Active up-packages whose home processes are in $T(p)$ could still exist at processes above p.

4. $p.status \in [0..4]$.

   Status variables are used to control the order of computation and to correct errors.

Finally, r contains the following additional variable:

5. $r.counter \in \mathbb{N}$.

   This incrementing integer variable assigns the rank to packages. It is initialized to be 0 every time a new epoch begins.

11.3.2 Predicates of $\mathcal{CRK}$

The predicate $\texttt{CleanState}(p)$ below indicates that p is in a good ("clean") initial state.

$$\texttt{CleanState}(p) \equiv p.up\_pkg = \perp \wedge p.down\_pkg = \perp \wedge \neg p.started \wedge \neg p.up\_done$$

The following four predicates are used for error detection:

$$\texttt{IsConsistent}(p, g) \equiv g = p.guide \vee (\exists q \in \texttt{Children}(p), g \geq q.guide)$$

$$\begin{aligned}\texttt{GuideError}(p) \equiv\ & (p.up\_pkg \neq \perp \wedge \neg\texttt{IsConsistent}(p, p.up\_pkg.guide)) \vee \\ & (p.down\_pkg \neq \perp \wedge \neg\texttt{IsConsistent}(p, p.down\_pkg.guide))\end{aligned}$$

$$\begin{aligned}\texttt{StatusError}(p) \equiv\ & (p.status \in \{1,3\} \wedge p.parent.status \neq p.status) \vee \\ & (p.status \in \{2,4\} \wedge (\exists q \in \texttt{Children}(p), q.status \neq p.status)) \vee \\ & (p.status \neq 0 \wedge p.parent.status = 0) \vee \\ & (p.status \notin \{0,1\} \wedge (\exists q \in \texttt{Children}(p), q.status = 0))\end{aligned}$$

$$\begin{aligned}\texttt{Error}(p) \equiv\ & \texttt{StatusError}(p) \vee \\ & (\neg\texttt{CleanState}(p) \wedge p.status = 1) \vee \\ & (\texttt{GuideError}(p) \wedge p.status = 2) \vee \\ & (p.up\_done \wedge \neg p.started \wedge p.status = 2) \vee \\ & (p.up\_done \wedge p.status = 2 \wedge (\exists q \in \texttt{Children}(p), \neg q.up\_done))\end{aligned}$$

We say that a guide pair g is *consistent with* p if $\texttt{IsConsistent}(p, g)$ is TRUE. If $\texttt{IsConsistent}(p, g)$ is FALSE, g is the guide pair of no process in the subtree of p. $\texttt{GuideError}(p)$ = TRUE means that p holds a package whose home is not in the subtree of p. The predicate $\texttt{StatusError}(p)$ indicates that p detects that its status is inconsistent with those of its neighbors. Status errors are always the result of arbitrary initialization; eventually, $\texttt{StatusError}(p)$ will become FALSE and will remain FALSE forever for all p. Finally, the predicate $\texttt{Error}(p)$ detects error in the current wave.

The following four predicates are used for flow control:

$$\begin{aligned}
\texttt{UpRedundant}(p) \quad &\equiv \quad (p \neq r \ \wedge \ p.up_pkg \neq \perp \ \wedge \ p.parent.up_pkg \neq \perp \wedge \\
&\quad\ p.parent.up_pkg \geq p.up_pkg) \vee (p = r \ \wedge \ p.up_pkg \neq \perp \wedge \\
&\quad\ p.down_pkg \neq \perp \ \wedge \ p.down_pkg.guide = p.up_pkg.guide)
\end{aligned}$$

$$\begin{aligned}
\texttt{DownReady}(p) \quad &\equiv \quad p.down_pkg \neq \perp \Rightarrow ((p.down_pkg.guide \neq p.guide \wedge \\
&\quad\ (\exists q \in \texttt{Children}(p), q.down_pkg = p.down_pkg)) \vee \\
&\quad\ (p.down_pkg.guide = p.guide \ \wedge \ p.rank = p.down_pkg.value))
\end{aligned}$$

$$\begin{aligned}
\texttt{CanStart}(p) \quad &\equiv \quad \neg p.started \ \wedge \ (p.up_pkg = \perp \ \vee \ \texttt{UpRedundant}(p)) \wedge \\
&\quad\ (\forall q \in \texttt{Children}(p), (q.up_done \vee \\
&\quad\ (\neg\texttt{UpRedundant}(q) \ \wedge \ q.up_pkg > (p.weight, p.guide))))
\end{aligned}$$

$$\begin{aligned}
\texttt{CanCopyUp}(p, q) \quad &\equiv \quad q \in \texttt{Children}(p) \ \wedge \ (q.up_pkg \neq \perp \ \wedge \ \neg\texttt{UpRedundant}(q)) \wedge \\
&\quad\ (p.up_pkg = \perp \ \vee \ \texttt{UpRedundant}(p)) \wedge \\
&\quad\ (p.started \ \vee \ (p.weight, p.guide) > q.up_pkg) \wedge \\
&\quad\ (\forall q' \in \texttt{Children}(p), (q'.up_done \ \vee \ (q'.up_pkg \neq \perp \wedge \\
&\quad\ \neg\texttt{UpRedundant}(q') \ \wedge \ q'.up_pkg \geq q.up_pkg)))
\end{aligned}$$

$p.up\_pkg$ is redundant if $\texttt{UpRedundant}(p)$ is TRUE. $\texttt{DownReady}(p)$ states that $p.down\_pkg$ is redundant or undefined, and thus p is permitted to create or copy a new down-package. $\texttt{CanStart}(p)$ states that p can create its own package, that is, p can set $p.up\_pkg$ to $(p.weight, p.guide)$. $\texttt{CanCopyUp}(p, q)$ states that p can copy $q.up\_pkg$ to $p.up\_pkg$. We note that p can evaluate $\texttt{UpRedundant}(q)$ for any $q \in \texttt{Children}(p)$.

Predicate $\texttt{UpDone}(p)$ below indicates that all processes in $T(p)$ have created their own up-package in the current epoch and that $T(p)$ contains no active up-package. The evaluation of $\texttt{UpDone}(p)$ gives the correct value for $p.up\_done$.

11.3.3 Actions of $\mathcal{CRK}$

Actions of $\mathcal{CRK}$ for the root process r are given in Algorithm 7. Actions of $\mathcal{CRK}$ for every non-root process p are given in Algorithm 8.

They achieve three tasks which are (1) error correction, (2) control of epochs, and (3) rank computation (using the flow of packages).

Algorithm 7 $\mathcal{CRK}$, code for the root process r only

Actions:

(1) `Err`	:: `Error`(r)	$\longrightarrow$	$r.status \leftarrow 0$
(2) `NewEpoch`	:: $r.status \in \{0,4\} \wedge$ $(\forall q \in$ `Children`$(r),$ $q.status \in \{0,4\})$	$\longrightarrow$	$r.status \leftarrow 1$; $r.up\_pkg \leftarrow \perp$; $r.down\_pkg \leftarrow \perp$; $r.started \leftarrow$ FALSE; $r.up\_done \leftarrow$ FALSE; $counter \leftarrow 0$
(3) `ConvCast`	:: $r.status = 1 \wedge$ $(\forall q \in$ `Children`$(r),$ $q.status = 2)$	$\longrightarrow$	$r.status \leftarrow 2$
(4) `CreateUpPkg`	:: $r.status = 2 \wedge$ `CanStart`(r)	$\longrightarrow$	$r.up\_pkg.value \leftarrow r.weight$; $r.up\_pkg.guide \leftarrow r.guide$; $r.started \leftarrow$ TRUE
(5) `CopyUpPkg`	:: $r.status = 2 \wedge$ $(\exists q \in$ `Children`$(r),$ `CanCopyUp`$(r,q))$	$\longrightarrow$	$r.up\_pkg \leftarrow q.up\_pkg,$ $q = min_{\prec_r}\{q' \in$ `Children`$(r),$ `CanCopyUp`$(r,q')\}$
(6) `EndUpPkg`	:: $r.started \wedge$ `UpRedundant`$(r) \wedge$ $(\forall q \in$ `Children`$(r),$ $q.up\_done)$	$\longrightarrow$	$r.up\_done \leftarrow$ TRUE
(7) `CreateDownPkg`	:: `DownReady`$(r) \wedge$ $r.up\_pkg \neq \perp \wedge$ $\neg$`UpRedundant`(r)	$\longrightarrow$	$counter \leftarrow counter + 1$; $r.down\_pkg.value \leftarrow counter$; $r.down\_pkg.guide \leftarrow r.up\_pkg.guide$
(8) `SetRank`	:: $r.down\_pkg \neq \perp \wedge$ $(r.down\_pkg.guide$ $= r.guide) \wedge$ $(r.down\_pkg.value$ $\neq r.rank)$	$\longrightarrow$	$r.rank \leftarrow r.down\_pkg.value$
(9) `BroadCast`	:: $r.status = 2 \wedge$ $r.up\_done \wedge$ `DownReady`(r)	$\longrightarrow$	$r.status \leftarrow 3$
(10) `EndEpoch`	:: $r.status = 3 \wedge$ $(\forall q \in$ `Children`$(r),$ $q.status = 4)$	$\longrightarrow$	$r.status \leftarrow 4$

Algorithm 8 $\mathcal{CRK}$, code for every non-root process p only

Actions:

(1) `Err`	:: `Error`(p)	$\longrightarrow$	$p.status \leftarrow 0$
(2) `NewEpoch`	:: $p.parent.status = 1 \wedge$ $p.status \in \{0,4\} \wedge$ $(\forall q \in$ `Children`$(p),$ $q.status \in \{0,4\})$	$\longrightarrow$	$p.status \leftarrow 1;$ $p.up\_pkg \leftarrow \perp;$ $p.down\_pkg \leftarrow \perp;$ $p.started \leftarrow$ FALSE; $p.up\_done \leftarrow$ FALSE
(3) `ConvCast`	:: $p.status = 1 \wedge$ $(\forall q \in$ `Children`$(p),$ $q.status = 2)$	$\longrightarrow$	$p.status \leftarrow 2$
(4) `CreateUpPkg`	:: $p.status = 2 \wedge$ `CanStart`(p)	$\longrightarrow$	$p.up\_pkg.value \leftarrow p.weight;$ $p.up\_pkg.guide \leftarrow p.guide;$ $p.started \leftarrow$ TRUE
(5) `CopyUpPkg`	:: $p.status = 2 \wedge$ $(\exists q \in$ `Children`$(p),$ `CanCopyUp`$(p,q))$	$\longrightarrow$	$p.up\_pkg \leftarrow q.up\_pkg,$ $q = \min_{\prec_p}\{q' \in$ `Children`$(p),$ `CanCopyUp`$(p,q')\}$
(6) `EndUpPkg`	:: $p.started \wedge$ `UpRedundant`$(p) \wedge$ $(\forall q \in$ `Children`$(p),$ $q.up\_done)$	$\longrightarrow$	$p.up\_done \leftarrow$ TRUE
(7) `CopyDownPkg`	:: `DownReady`$(p) \wedge$ $(p.parent.down\_pkg \neq \perp) \wedge$ $(p.parent.down\_pkg \neq p.down\_pkg) \wedge$ `IsConsistent`$(p,$ $p.parent.down\_pkg)$	$\longrightarrow$	$p.down\_pkg \leftarrow p.parent.down\_pkg$
(8) `SetRank`	:: $p.down\_pkg \neq \perp \wedge$ $p.down\_pkg.guide = p.guide) \wedge$ $(p.down\_pkg.value \neq p.rank)$	$\longrightarrow$	$p.rank \leftarrow p.down\_pkg.value$
(9) `BroadCast`	:: $p.parent.status = 3 \wedge$ $p.status = 2 \wedge$ $(\forall q \in$ `Children`$(p),$ $q.status = 2) \wedge$ `DownReady`(p)	$\longrightarrow$	$p.status \leftarrow 3$
(10) `EndEpoch`	:: $p.status = 3 \wedge$ $(\forall q \in$ `Children`$(p),$ $q.status = 4)$	$\longrightarrow$	$p.status \leftarrow 4$

Error Correction. Action `Err` performs the error correction. If one process detects any inconsistency among its state and that of its neighbors, it initiates a reset of the network by changing its status to 0.

Epochs. We now describe what happens during one epoch. In this description, we assume that the epoch contains no initialization errors. (As mentioned above, if any process detects such an error, the epoch is aborted, and a new, error-free, epoch begins.)

The new epoch starts when r executes Action `NewEpoch`. If $r.status$ is either 0 or 4, and every child of r has status 0 or 4, then r broadcasts a status 1 wave and resets to a clean state.

When the status 1 wave reaches the leaves, all processes execute Action `ConvCast` in a convergecast wave, changing status to 2, so that rank computation can begin.

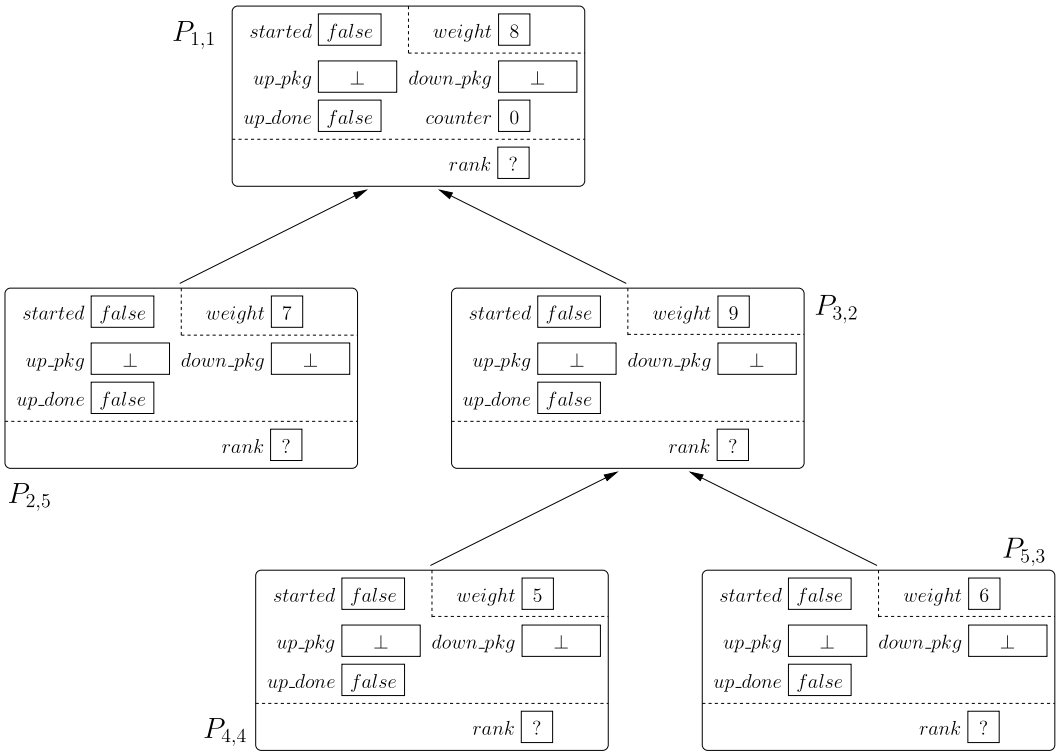
When r detects that there are no more up-packages in the tree, and it has already sent every down-package, it initializes a status 3 broadcast wave by executing Action `BroadCast`. Note that there could still be active down-packages below r, but there could not be any active up-packages. Thus, r is finished with its task for the current epoch. A non-root process p propagates the status 3 wave by Action `BroadCast` after sending all its down-packages. There could still be active down-packages below p, but no active up-packages. Since $p.parent.status = 3$ and the down-package at p is redundant, p knows that its job for this epoch is done, and consequently changes its status to 3.

Once the status 3 wave reaches the leaves, all process execute Action `EndEpoch` in a convergecast status 4 wave. When that wave reaches r, the current epoch is done, and r initiates a new epoch.

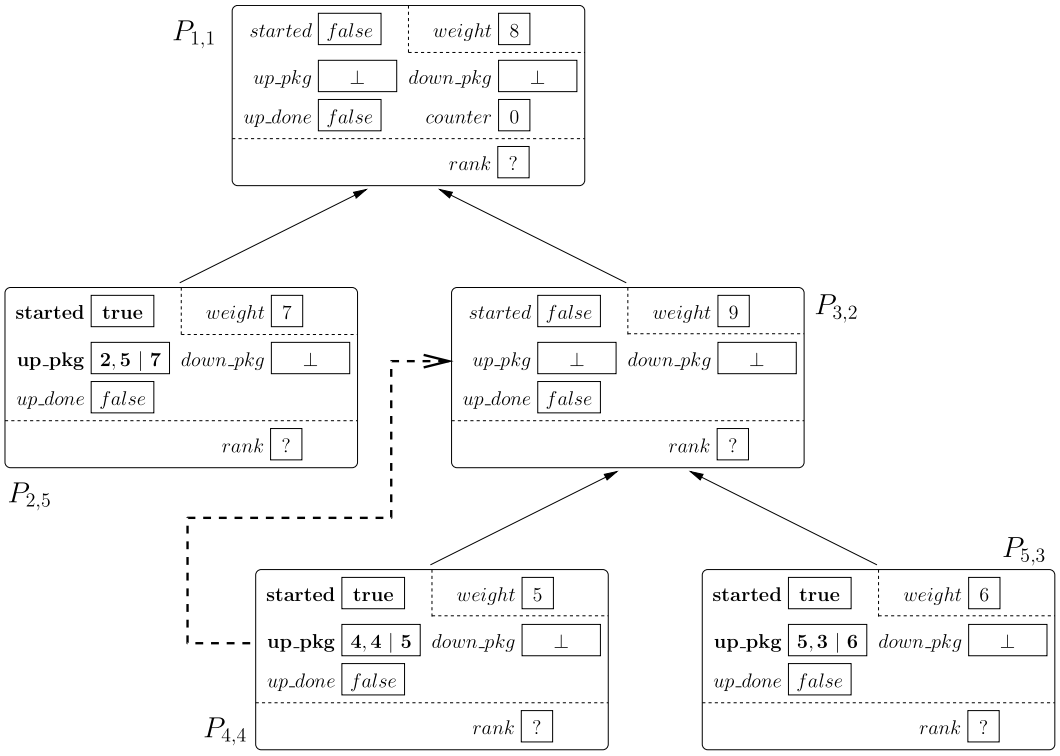
Rank Computation. The computation of the ranks is bottom-up, and starts when the convergecast status 2 wave starts at the leaves. The flow of up-packages is organized using `CreateUpPkg` and `CopyUpPkg`, that is, a process either inserts its own package in the flow or copies some package coming from a child in such a way to ensure that packages are moved up in ascending order of weight. Once a process p has detected that $T(p)$ has no active up-package, it sets $p.up\_done$ to TRUE by Action `EndUpPkg`. r initializes the broadcast of the status 3 wave only after $r.up\_done$ changes to TRUE.

When r receives a new up-package, that is, $r.up\_pkg$ becomes active, if $r.down\_pkg$ is available (that is, it is either $\perp$ or redundant), r is enabled to create a new down-package by executing `CreateDownPkg`. If $counter = \rho$, then $r.up\_pkg$ is the ρ^{th} up-package copied or created by r, *i.e.*, its weight is the ρ^{th} smallest weight in the network; ρ will then become the value of the down-package.

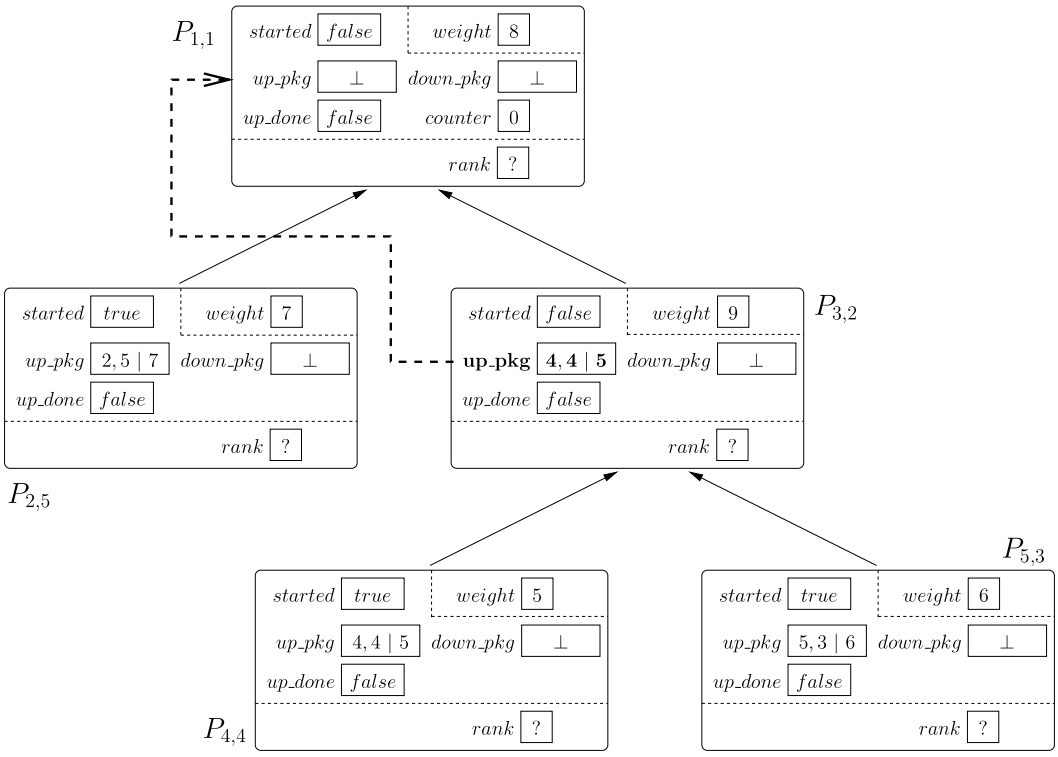
The new active down-package is propagated to its home process by forward copying, guided by its guide pair, using Action `CopyDownPkg`. When it reaches its home process p, the value field of that package contains the correct value of the rank of p. p updates $p.rank$ using Action `SetRank`, if necessary.


$P_{1,1}$
started false
weight 8
up_pkg ⊥
down_pkg ⊥
up_done false
counter 0
rank ?
started false
weight 7
up_pkg ⊥
down_pkg ⊥
up_done false
rank ?
$P_{2,5}$
started false
weight 9
up_pkg ⊥
down_pkg ⊥
up_done false
rank ?
$P_{3,2}$
started false
weight 5
up_pkg ⊥
down_pkg ⊥
up_done false
rank ?
$P_{4,4}$
started false
weight 6
up_pkg ⊥
down_pkg ⊥
up_done false
rank ?
$P_{5,3}$

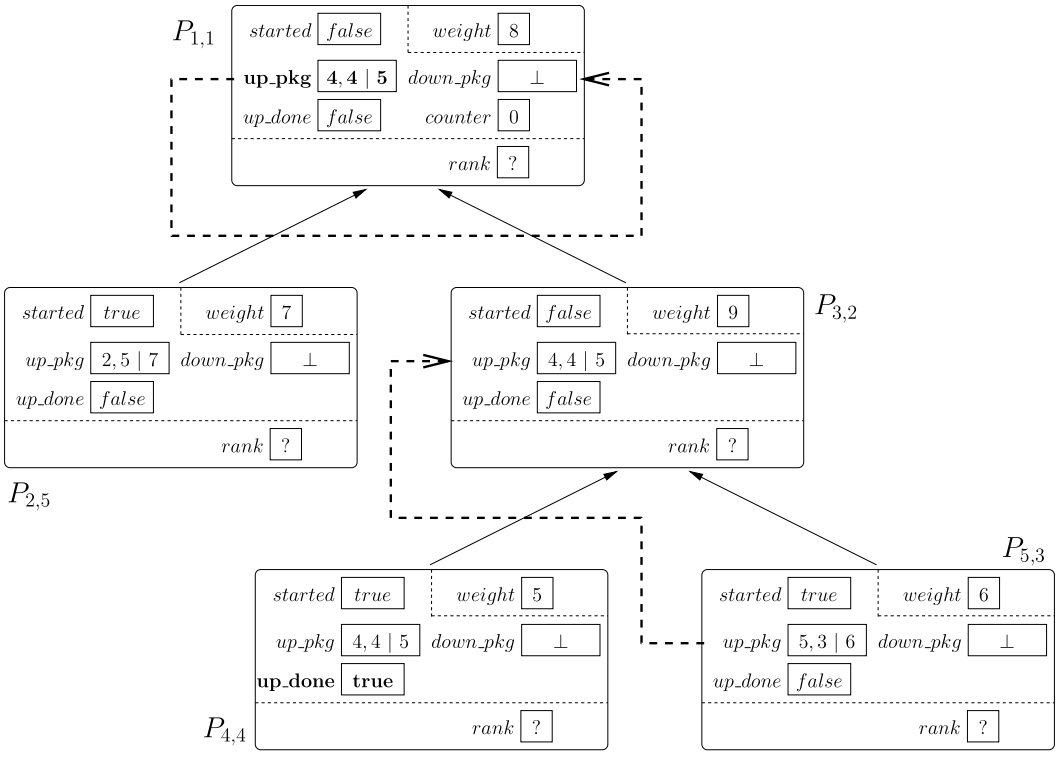
(a)


$P_{1,1}$
started false
weight 8
up_pkg ⊥
down_pkg ⊥
up_done false
counter 0
rank ?
started true
weight 7
up_pkg 2, 5 | 7
down_pkg ⊥
up_done false
rank ?
$P_{2,5}$
started false
weight 9
up_pkg ⊥
down_pkg ⊥
up_done false
rank ?
$P_{3,2}$
started true
weight 5
up_pkg 4, 4 | 5
down_pkg ⊥
up_done false
rank ?
$P_{4,4}$
started true
weight 6
up_pkg 5, 3 | 6
down_pkg ⊥
up_done false
rank ?
$P_{5,3}$

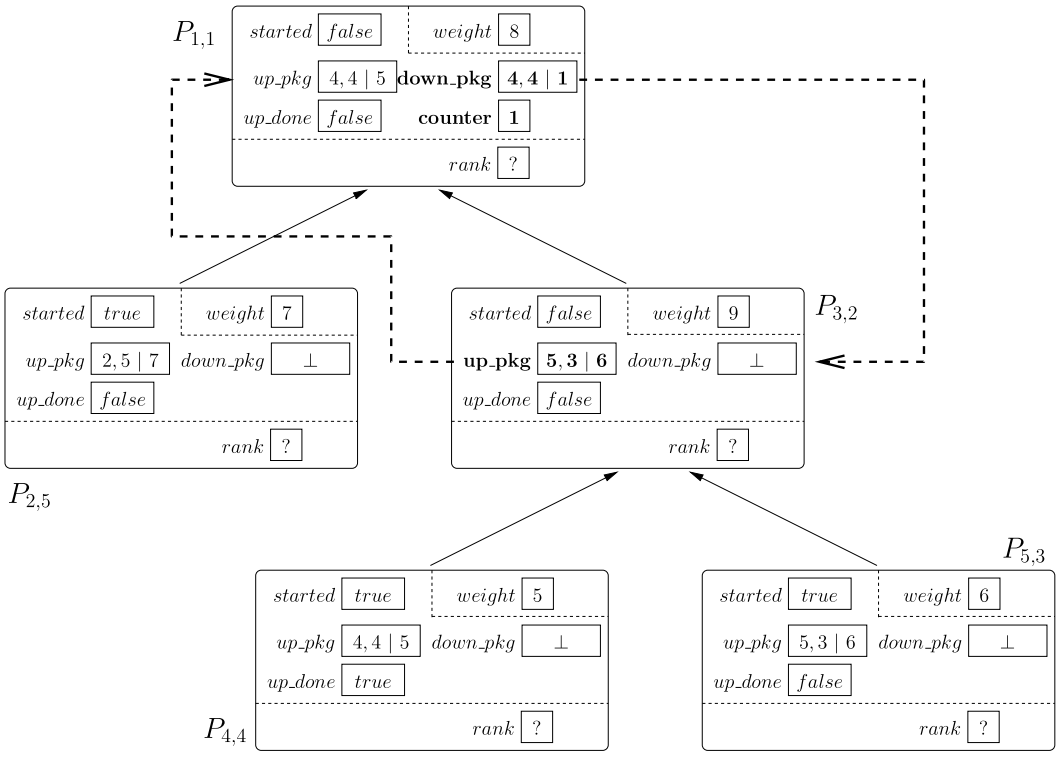
(b)


$P_{1,1}$
started false
weight 8
up_pkg ⊥
down_pkg ⊥
up_done false
counter 0
rank ?
started true
weight 7
up_pkg 2, 5 | 7
down_pkg ⊥
up_done false
rank ?
$P_{2,5}$
started false
weight 9
$P_{3,2}$
up_pkg 4, 4 | 5
down_pkg ⊥
up_done false
rank ?
$P_{5,3}$
started true
weight 5
up_pkg 4, 4 | 5
down_pkg ⊥
up_done false
rank ?
$P_{4,4}$
started true
weight 6
up_pkg 5, 3 | 6
down_pkg ⊥
up_done false
rank ?

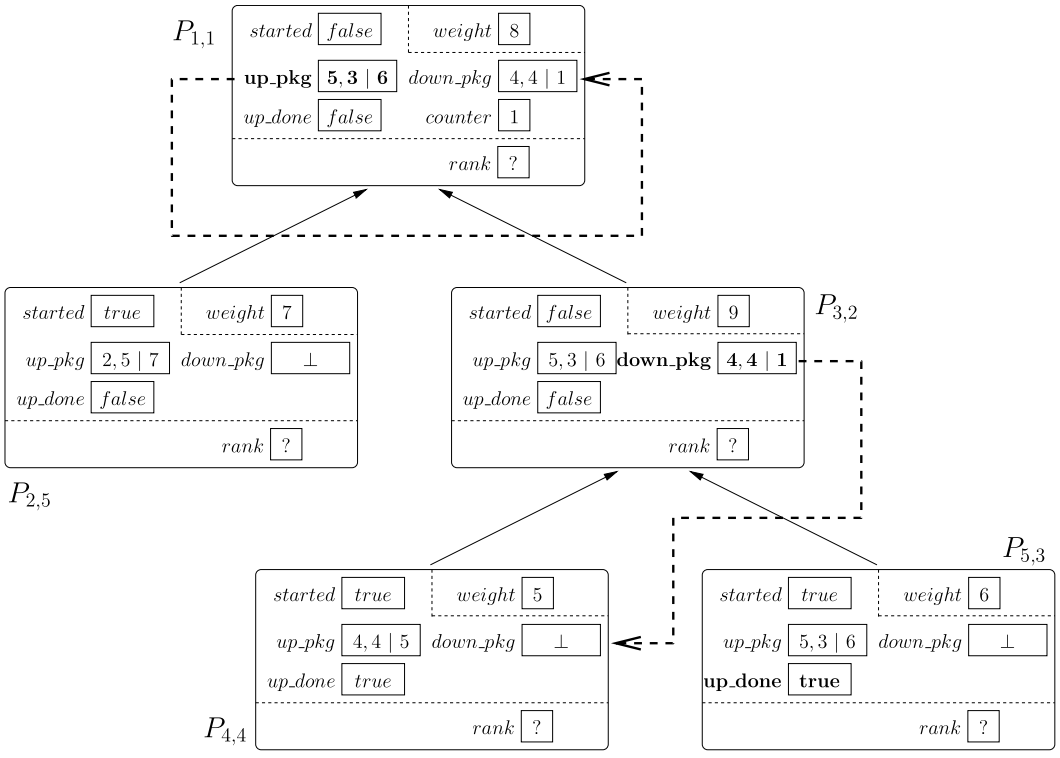
(c)


$P_{1,1}$
started false
weight 8
up_pkg 4, 4 | 5
down_pkg ⊥
up_done false
counter 0
rank ?
started true
weight 7
up_pkg 2, 5 | 7
down_pkg ⊥
up_done false
rank ?
$P_{2,5}$
started false
weight 9
$P_{3,2}$
up_pkg 4, 4 | 5
down_pkg ⊥
up_done false
rank ?
$P_{5,3}$
started true
weight 5
up_pkg 4, 4 | 5
down_pkg ⊥
up_done true
rank ?
$P_{4,4}$
started true
weight 6
up_pkg 5, 3 | 6
down_pkg ⊥
up_done false
rank ?

(d)


$P_{1,1}$
started false
weight 8
up_pkg 4, 4 | 5
down_pkg 4, 4 | 1
up_done false
counter 1
rank ?
started true
weight 7
up_pkg 2, 5 | 7
down_pkg ⊥
up_done false
rank ?
$P_{2,5}$
started false
weight 9
$P_{3,2}$
up_pkg 5, 3 | 6
down_pkg ⊥
up_done false
rank ?
$P_{5,3}$
started true
weight 5
up_pkg 4, 4 | 5
down_pkg ⊥
up_done true
rank ?
$P_{4,4}$
started true
weight 6
up_pkg 5, 3 | 6
down_pkg ⊥
up_done false
rank ?

(e)


$P_{1,1}$
started false
weight 8
up_pkg 5, 3 | 6
down_pkg 4, 4 | 1
up_done false
counter 1
rank ?
started true
weight 7
up_pkg 2, 5 | 7
down_pkg ⊥
up_done false
rank ?
$P_{2,5}$
started false
weight 9
$P_{3,2}$
up_pkg 5, 3 | 6
down_pkg 4, 4 | 1
up_done false
rank ?
$P_{5,3}$
started true
weight 5
up_pkg 4, 4 | 5
down_pkg ⊥
up_done true
rank ?
$P_{4,4}$
started true
weight 6
up_pkg 5, 3 | 6
down_pkg ⊥
up_done true
rank ?

(f)

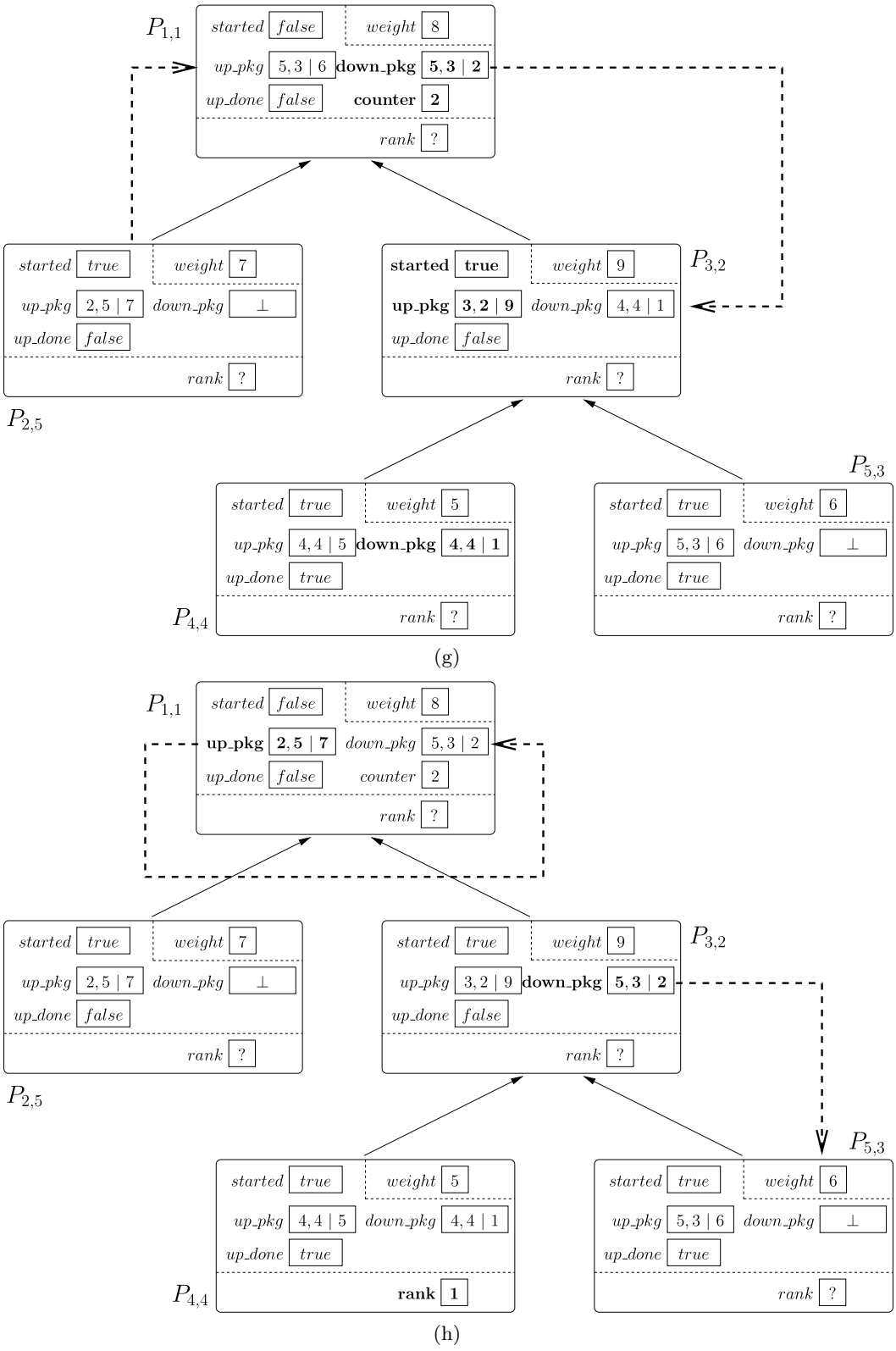


Figure 11.4 – Example of an execution until the first rank is assigned.

Figure 11.4 depicts a synchronous execution of a rank computation. For every process p, we show its inputs (processes are subscripted with their guide pair and their weight is given upper right), some of its computation variables (in the middle: up-package, down-package, *up_done* and *started* flags and root-counter) and its output (at the bottom: rank). At each step, when the value of a variable changes, we write the new value in bold. Dashed arrows show the next moves of a up- or down-package.

The example starts in a configuration where every computation variable has been reset by Action `NewEpoch` (Figure 11.4a). The output variables *rank* hold arbitrary values, denoted by "?". In Figure 11.4b, every leaf creates its own up-package with its guide pair and its weight. The up-packages are then routed, in weight order, up to the root, as shown in Figures 11.4c and 11.4d. In Figure 11.4e, the root process $r = p_{1,1}$ increments its counter to 1 and creates the first down-package of the current epoch: the smallest weight is 5 and is held by the process labeled by the guide pair $(4, 4)$. This down-package is routed down to $p_{4,4}$, thanks to guide pairs, as shown in Figures 11.4f and 11.4g. Finally, in Figure 11.4h, $p_{4,4}$ assigned its own rank.

11.4 Correctness of Algorithm $\mathcal{CRK}$

By Corollary 1 (page 37), to show the correctness of $\mathcal{RANK}$, it suffices to show that the variables of $\mathcal{CRK}$ stabilize to their correct values, starting from any silent legitimate configuration of $\mathcal{GUIDE}$. Let γ be such a configuration. The first part of the proof deals with error correction.

We say that a process p is *inconsistent* if $p.status = 2$, $p.up\_done$, and there is some $q \in \texttt{Children}(p)$ such that $q.up\_done = \text{FALSE}$.

Lemma 48 *If at least one round has elapsed after configuration γ, the following conditions hold for every process p:*

(a) $\neg\texttt{CleanState}(p) \wedge (p.\text{status} = 1)$ *is* FALSE.

(b) $p.\text{up\_done} \wedge \neg p.\text{started} \wedge (p.\text{status} = 2)$ *is* FALSE.

(c) *p is not inconsistent; i.e., if* $p.\text{status} = 2$ *and* $p.\text{up\_done} = \text{TRUE}$, *then* $q.\text{up\_done} = \text{TRUE}$ *for all* $q \in \texttt{Children}(p)$.

Proof. Let consider the three conditions separately.

(a) If $p.status = 1$ and $\texttt{CleanState}(p) = \text{FALSE}$, then p is enabled to execute Action `Err`. Moreover, this condition only deals with local variables of p. So, Action `Err` is continuously enabled, and p executes $p.status \leftarrow 0$ in at most one round. Then, $\neg\texttt{CleanState}(()p) \wedge (p.status = 1)$ is FALSE.

Assume $\neg\texttt{CleanState}(()p) \wedge (p.status = 1)$ is FALSE. Then, if $p.status = 1$, p cannot modify its other variables before changing its status. Moreover,

every time $p.status$ is reset to 1, the other variables are reset to a *clean* state (see Action `NewEpoch`). So, $\neg$`CleanState`$(()p) \wedge (p.status = 1)$ remains FALSE forever.

(b) If $p.up\_\ done$ = TRUE, $p.started$ = FALSE, and $p.status = 2$, then p is enabled to execute Action `Err`. Moreover, this condition only deals with local variables of p. So, Action `Err` is continuously enabled, and p executes $p.status \leftarrow 0$ in at most one round. Then, $p.up\_\ done \wedge \neg p.started \wedge (p.status = 2)$ is FALSE.

Assume $p.up\_\ done \wedge \neg p.started \wedge (p.status = 2)$ is FALSE. Then, p always sets $p.up\_\ done$ and $p.started$ to FALSE together in Action `NewEpoch`. Moreover, p sets $p.up\_\ done$ to TRUE only if $p.started$ holds (see Action `EndUpPkg`). So, $p.up\_\ done \wedge \neg p.started \wedge (p.status = 2)$ remains FALSE forever.

(c) Assume p is inconsistent. Then, in one round, either every $q \in$ `Children`$(()p)$ satisfies $q.up\_\ done$ = TRUE or p executes Action `Err`. In both cases, p is no more inconsistent.

Assume p is not inconsistent. Then, p sets $p.up\_\ done$ to TRUE, by executing Action `EndUpPkg`, only when every $q.up\_\ done$ = TRUE for all $q \in$ `Children`(p). Moreover, any $q \in$ `Children`(p) sets $q.up\_\ done$ to FALSE, by executing Action `NewEpoch`, only when $p.up\_\ done$ = FALSE. Thus, p cannot later become inconsistent.

□

Lemma 49 *If at least one round has elapsed after configuration γ, and if* `StatusError`(p) = TRUE*, then one of the following conditions holds:*

- $p \neq r$ *and* p.parent.status $= 0$.
- *There is some* $q \in$ `Children`(p) *such that* q.status $= 0$.

Proof. First, values 1 and 3 are propagated in the tree by broadcast waves. Then, values 2 and 4 are propagated in the tree by convergecast waves. So, by definition of `StatusError`(p), if `StatusError`(p) = FALSE at some point, then `StatusError`(p) will become TRUE only after some neighbor of p switches its status to 0. Finally, by the definition of Action `Err`, p cannot satisfy `StatusError`(p) = TRUE during one round without changing its status to 0. □

Lemma 50 *If a process with status 0 holds an active package, this package remains blocked until it is removed or cleaned.*

Proof. If a process p has status 0, then no other process can copy its up or down packages because each of its neighbors either has status 0, is its parent and has status 1, or is enabled to execute Action `Err`, the action with the highest priority. The next time p changes its status by executing Action `NewEpoch`, its state will become clean.

□

Lemma 51 *Within $O(n)$ rounds from γ, if process p contains an active package such that there is no process in its subtree which is the home process of that package, then p.*status $= 0$.

Proof. Consider any configuration γ' after one round from γ. Consider an active package x in γ' at any process p such that there is no process in the subtree of p that is the home process of that package.

Assume that there is an ancestor of p with status 0, or a process in the subtree of p with status 0. Then, in at most h rounds, any process that holds x as an active package has status 0 by Action `Err` (remember that processes with status 1 do not hold any package, by Lemma 48), and by Lemma 50, x cannot be copied anymore, so we are done.

Assume that no ancestor and no descendant of p have status 0. We have four cases, depending on the status of p.

(a) $p.status = 4$. Assume that there is an ancestor q of p whose status is 1. By Lemma 49 and the definition of `StatusError`, all descendants of p have status 4, and for every ancestor q of p, we have $q.status \in \{1, 4\}$ and $(q.status = 1) \Rightarrow (q = r) \vee (q.parent.status = 1)$. Thus, in at most h rounds, the subtree of p has been reset to a clean state by Action `NewEpoch`, and we are done.

Assume that there is no ancestor q' of p such that $q'.status = 1$. Then, by Lemma 49 and the definition of `StatusError`, all descendants of p have status 4, and for every ancestor q of p we have $q.status \in \{3, 4\}$ and $(q.status = 3) \Rightarrow (q = r) \vee (q.parent.status = 3)$. Thus, in at most h rounds, all ancestors of p will change to status 4 by executing Action `EndEpoch`, and we reduce to the previous case.

(b) $p.status = 3$. If there is a process that has status 4, we reduce to the previous case, by Lemma 49 and the definition of `StatusError`.

Otherwise, every process of the tree has status 2 or 3, and if a process has status 3, then either it is r, or its parent also has status 3, by Lemma 49 and the definition of `StatusError`. In this case, x can only be copied down in the tree (and only if it is a down package). In $O(n)$ rounds, one of the following conditions will hold.

(i) x becomes an active package of a process q such that `GuideError`$(q) \wedge (q.status = 2)$. (In the worst case q is a leaf.) The children of q cannot copy x, and after one additional round, q has status 0, and x cannot be copied anymore, by Lemma 50, so we are done.

(ii) The broadcast wave of status 3 reaches the leaves of the tree, and in at most h additional rounds, after the convergecast of the status 4 wave, we have Case (a).

(c) $p.status = 2$. If there is a process q such that $q.status = 3$, then $r.status = 3$ by Lemma 49 and the definition of `StatusError`, and we reduce to Case (b).

Otherwise, by Lemma 49, every process has status 1 or 2, and if a process has status 1, either it is r or its parent has status 1. By executing Action `ConvCast`, all processes of T have status 2 within at most h rounds.

- If x is an up-package, it can only be copied up the tree. Either p satisfies $\texttt{GuideError}(p) \wedge (p.status = 2)$, its parent cannot copy x, after one round p has status 0, and x cannot be copied any more (by Lemma 50), so we are done; or in $O(n)$ rounds, x becomes a down package at the r, which has status 2, and is no longer an active up-package at any process.
- If x is a down-package, it can only be copied down in the tree. After $O(n)$ rounds, the host q of x satisfies $\texttt{GuideError}(q) \wedge (q.status = 2)$. (In the worst case q is a leaf.) The children of q cannot copy x. After one additional round, q has status 0, and, by Lemma 50, x cannot be copied anymore; hence we are done.

(d) $p.status = 1$. By Lemma 48, p does not hold any package, so this case is contradictory.

□

Lemma 52 *Within $O(n)$ rounds from γ, if a process p contains a package, then there is a process in its subtree which is the home process of that package.*

Proof. By Lemmas 50 and 51, after $O(n)$ rounds, every process p holding an active package that does not have its home in the subtree of p satisfies $p.status = 0$ and no process copies this package.

The status 0 wave is propagated in $O(h)$ rounds, by Action `Err`, up the tree until reaching the root, or a process with status 1 all of whose ancestors also have status 1, causes all processes in the subtree $T(p)$ to change their status to 1 within $O(h)$ rounds by executing Action `NewEpoch`.

Hence, within $O(n)$ rounds, all inconsistent active packages will be removed from the tree, by Lemma 49. □

By Lemmas 48, 49, and 52, within $O(n)$ rounds from γ, $\texttt{Error}(p)$ is FALSE forever for each process p. There may still exist processes with status 0, but in that case, by the definition of `Error`, and for any process p, we have $p.status\{0, 1, 4\}$, $(p.status = 0) \Rightarrow (p = r) \ \vee \ (p.parent.status \in \{0, 1\})$, $(p.status = 1) \Rightarrow (p = r) \vee (p.parent.status = 1)$, and $(p.status = 4) \Rightarrow (p \neq r) \ \wedge \ (p.parent.status \in \{1, 4\})$. Hence, at the end of the status 1 broadcast wave, which takes at most $O(h)$ rounds, no process will have status 0. Thus, we have the following lemma:

Lemma 53 *Within $O(n)$ rounds after configuration γ, $\texttt{Error}(p)$ is* FALSE *and* $p.\text{status} \in \{1, 2, 3, 4\}$ *forever, for each process p.*

From Lemma 53, we can deduce that the following invariant holds within $O(n)$ rounds after γ for all p.

1. $\texttt{Error}(p)$ is FALSE and $p.status \in \{1, 2, 3, 4\}$.

   That is, all initial errors will eventually be corrected.

2. If $p.status \in \{1, 3\}$, then either $p = r$ or $p.parent.status = p.status$.

3. If $p.status \in \{2, 4\}$, then $q.status = p.status$ for all $q \in \texttt{Children}(p)$.

We now show that, starting from any configuration where all previous invariants hold, infinitely many complete epochs are executed, and each of those epochs takes $O(n)$ rounds.

- If $r.status = 4$, then all processes have status 4 and r initiates a status 1 broadcast wave by executing Action `NewEpoch`.

- If $r.status = 1$, then all processes p have either status 1 or 4. Moreover, $(p.status = 1) \Rightarrow (p = r) \vee (p.parent.status = 1)$. Thus, the status 1 broadcast wave reaches all processes in at most h rounds.

- After the status 1 wave reaches the leaves, the status 2 convergecast wave is initiated by the leaves by execution of Action `ConvCast`, and moves to r in at most h rounds.

- Once $r.status = 2$, all processes have status 2. The flow of packages starts in parallel at processes of status 2.

- By Claim 1, for every process p, if $p.status = 2$ and $p.up\_done$, every process q in $T(p)$ subtree satisfies $q.up\_done$. Moreover, $\neg p.started \Rightarrow \neg p.up\_done$. By executing Action `CreateUpPkg`, the deepest process p satisfying $p.status = 2$ and $\neg p.started$ eventually sets $p.started$ to TRUE and initiates its own up-package. The up-packages go up in the tree in weight order. Every process p satisfies $p.up\_done$ after $O(n)$ rounds.

- When each process p satisfies $p.up\_done$, r eventually satisfies $\texttt{DownReady}(r)$. Then, r initiates the status 3 broadcast wave by executing Action `BroadCast`.

- When $r.status = 3$, all processes p have either status 2 or 3. Moreover, $(p.status = 3) \Rightarrow (p = r) \vee (p.parent.status = 3)$. So, status 3 is broadcast to the whole tree by Action `BroadCast`. As each process must wait for its down-package to become redundant before switching to status 3, this phase is takes $O(n)$ rounds.

- Finally, once the status 3 wave reaches a leaf, the status 4 convergecast wave is initiated. That wave is completed within at most h rounds. r eventually has status 4, again.

Consider now any epoch that starts from a configuration, where all previous invariants (1-3) hold. We define $S = \{q : q.status \in \{1, 2, 3\}\}$. We call S the *active portion* of T. The following invariants hold for all $p \in S$.

4. If $p.status = 1$, then $p.started$ and $p.up\_done$ are FALSE, and $p.up\_pkg = p.down\_pkg = \perp$.

   If the status of p is 1, then p has initialized its variables and has not yet begun the calculations of the epoch.

   Proof: By Claim 1, and definitions of `Error`(p) and `CleanState`(p). □

5. If $p.up\_done$ then $p.started$, and $q.up\_done$ for all $q \in$ `Children`(p).

   If there is no active up-package in $T(p)$, then there is no active up-package in $T(q)$ for any child q. Furthermore, the package whose home is p has already been created and copied up.

   Proof: $p.up\_done$ is initialized to FALSE for all processes p during the broadcast wave of status 1 (Claim 4). Then, all $p.up\_done$ are set to TRUE in a bottom up fashion by Action `EndUpPkg`. □

6. $p.up\_done$ if and only if there is no active up-package in $T(p)$.

   Proof: $p.up\_done$ is initialized to FALSE for all processes p during the broadcast wave of status 1 (Claim 4). Then, all $p.up\_done$ are set to TRUE in a bottom up fashion by Action `EndUpPkg`. We can verify this claim by induction. □

7. If p hosts an active up-package, there is some process $q \in T(p) \cap S$ that is the home process of that package.

   Proof: If there is no process $q \in T(p)$ that is the home process of that package, then in $O(n)$ rounds, some process q' satisfies $q'.status = 0$ by Lemma 51, a contradiction to Claim 1.

   Assume that $q \notin S$, that is, $q.status = 4$. Then, $r.status \in \{3, 4\}$ by Claims 1-3. Before satisfying $r.status \in \{3, 4\}$, r has changed its status from 1 to 2 and from 2 to 3. But, r changes its status to 3 only if $r.up\_done$ (see Action `BroadCast`). In this case, there is no active up-package in T by Claim 6, a contradiction. □

8. If $p.started$ is FALSE, then there is no active package in S whose home process is p.

   Proof: p resets $p.started$ to FALSE during the status 1 broadcast wave (Claim 4). Moreover, when p receives status 1 broadcast wave, all package variables in the path from p to r have been reset. Then, $p.started$ remains FALSE until p creates its up-package. □

9. If $p.started$ is TRUE, then there is at most one active package whose home process is p.

Proof: *p.started* switches to TRUE only if $p.status = 2$. Then, all descendants and all ancestors have reset their package variables before p switches *p.status* to 2. So, there is no active package whose home is p hosted by these processes. Moreover, there is no such package anywhere else, otherwise in $O(n)$ rounds, some process q will satisfy $q.status = 0$ by Lemma 51; a contradiction to Claim 1. Hence, when p is enabled to switches *p.started* to TRUE, there is no active package whose home process is p.

Then, p creates its own package only once (when switching *P.started* to TRUE), and once a package has been copied, previous copies become redundant. □

10. If $q \in$ `Children`(p) and if $p.up\_pkg \neq \perp$, then either $q.up\_pkg.weight \geq p.up\_pkg.weight$ or $q.up\_done$.

    Proof: Minimum heap-order is maintained, so that up-packages reach r in weight order. □

11. Let x be the number of processes p such that *p.started* is FALSE or $p \notin S$, and y be the number of active up-packages in S. If $r.status \in \{1, 2\}$, then $x + y + counter = n$, the size of T.

    Proof: At the start of each epoch, $x = n$ and $y = counter = 0$. Each time a process executes Action `CreateUpPkg`, x is decremented and y is incremented. Each time r executes Action `CreateDownPkg`, y is decremented and *counter* is incremented. At the end (that is, last configuration before r takes status 3), $x = y = 0$ and $counter = n$. □

12. If *p.weight* is the i^{th} smallest weight in T, then $i > counter$ if and only if either *p.started* is FALSE, or there is an active up-package whose home process is p.

    Proof: From Claims 7-11. □

13. If *p.started* is TRUE and *p.rank* is not the correct rank of p, then there is an active package in S whose home process is p.

    Proof: From the previous claim, the active up-package whose home is p will cause the creation of a down-package whose home is p with the correct rank value. □

14. If $p.status = 3$, then *p.started* and *p.up_done* are TRUE, *p.up_pkg* and *p.down_pkg* are both redundant, and *p.rank* has the correct value.

    If the status of p is 3, then p has completed its role in the epoch.

    Proof: r is the first process to change its status to 3 during the epoch (by Action `BroadCast`), hence when 3, *r.up_done* is TRUE, which, in turn, implies that *p.up_done* is TRUE (Claim 5). Moreover, if *p.up_done*, then *p.started* and *p.up_pkg* are redundant (see Action `EndUpPkg`). p gets status 3 only if there is no active down-package in the path from the root to p, so *p.down_pkg* is redundant, too (see Action `BroadCast`). Finally, by Claim 13, *p.rank* has the correct value. □

Infinitely many complete epochs are executed, and during each of these epochs, all processes switch to status 3. By Claim 14, we thus have the following theorem:

Theorem 26 *$\mathcal{RANK}$ is self-stabilizing, computes the ranking of all processes in $O(n)$ rounds from an arbitrary initial configuration, under a weakly fair daemon.*

Conclusion

In this thesis, we studied the property of self-stabilization applied to the construction of distributed data structures.

First, we introduced the motivation of our work by presenting the field of distributed systems and more specifically the state of research work on self-stabilization in Chapter 1. In the two other chapters of Part I, we defined the theoretical tools we used for modeling distributed systems and reasoning on them in the following parts of this thesis.

Then, in Part II, we presented both the notions of maximum independent set (MIS) tree and guide pairs, which have been found to be very useful to some of our works. We first gave a silent self-stabilizing algorithm that finds an MIS tree of any network under a weakly fair daemon in Chapter 4. After proving its correctness and its linear convergence in rounds, we showed that the problem it solves is $\mathcal{P}$-complete. This spared us seeking an MIS tree construction in sublinear time. We used this algorithm in our competitive k-clustering construction presented in the next part of this thesis. In Chapter 5, we detailed the notion of guide pairs which are a special labeling of tree networks. We later made use of guide pairs for solving the ranking problem in Chapter 11. We described a silent self-stabilizing algorithm that computes guide pairs in any tree network and proved its correctness under a weakly fair daemon. We also showed it converges in a linear number of rounds with respect to the height of the tree network.

Afterwards, in Part III, we studied the problem of k-clustering. We introduced the notions of k-clustering and k-dominating set in Chapter 6, then exposed a possible application of k-clustering, and proposed two approaches for seeking optimization. In Chapter 7, we fixed a proof that establishes an upper bound on the size of the minimum k-dominating set with respect to the size of the network. Inspired by the scheme of this proof, we proposed a silent self-stabilizing algorithm that finds a minimal k-dominating set of bounded size in any network. After proving its correctness, we showed that it converges in a linear number of rounds under a weakly fair daemon. In Chapter 8, we proposed a silent self-stabilizing algorithm that computes a k-clustering of any tree network under a weakly fair daemon. We proved that it finds a minimum k-clustering in tree networks. By composing it with the MIS tree construction we previously gave, we obtained a more general solution for arbitrary networks. We established its correctness and its stabilization time which is linear in rounds. Moreover, we showed that our k-clustering is competitive when the communication topology of the distributed system is an unit-disk graph (UDG) or an approximate disk graph (ADG), which is a generalization of UDG. Such graphs are commonly used to model wireless ad hoc networks. Finally, we evaluated our algorithms through simulations. We presented and analyzed our experimentation results in Chapter 9.

Additionally, we studied the problem of constructing a minimal (f, g)-alliance of an arbitrary network in Part IV. This is an generalization of many spanning structure construction problems of interest in distributed systems. We proposed a self-stabilizing solution with safe convergence under an unfair daemon assuming $f \geq g$. Beyond proving its correctness, we showed that its first convergence time is four rounds at most and its second convergence time is linear in rounds with respect to the size of the network.

Finally, we broached the problem of ranking in tree networks in Part V. We proposed a self-stabilizing solution that converges in a linear number of rounds under a weakly fair daemon and proved its correctness. This is an application example showing the usefulness of guide pairs for navigating in tree networks.

Perspectives

Guide Pairs. Our work on self-stabilizing computation of *guide pairs* can be continued on several points. First, a direct extension of our work in Chapter 5 would be to prove that our solution $\mathcal{GUIDE}$ also works under an unfair daemon, that is, it does not require the assumption of a weakly fair daemon. Our fairness transformer given in Chapter 3 proves that a solution exists under an unfair daemon. However, the transformed algorithm $\mathcal{GUIDE}^t$ self-stabilizes in $O(\mathcal{D}n^3)$ steps, whereas we conjecture that our current solution $\mathcal{GUIDE}$ self-stabilizes in $O(nh)$ steps under an unfair daemon. Then, more applications of guide pairs should be studied, as we did with the distributed ranking algorithm in Chapter 11. For example, we thought about using this labeling for implementing a routing scheme over a clustering. Actually, a cluster is a connected set of processes in which one process is distinguished to be the clusterhead. Thus, it is possible to build a spanning tree of a cluster routed at its clusterhead. As a matter of fact, such a tree is already built by our algorithm $\mathcal{CLR}(k)$ described in Chapter 8. To achieve inter-cluster communication, processes must be able to communicate in the two following ways. First, any process may have to send a message to its clusterhead. This can be done using the parent pointers of the tree spanning its own cluster. Secondly, the other way of communication, that is, from a clusterhead to some member of its cluster, may also be useful. This can be carried out efficiently thanks to guide pairs.

k-Clustering. This brings us to the possible extensions of our work on the self-stabilizing construction of k-clustering presented in Part III. First, we think it is still possible to improve the stabilization time of our algorithm which is currently in $O(n)$ rounds. Ideally, we would like to propose a self-stabilizing solution that converges to a k-clustering in $O(k)$ rounds. Note that we showed in Chapter 4 that our MIS tree construction could not cope with this, so another approach has to be found. Besides, we would like to pursue the construction of *competitive* k-clustering in a larger class of network topologies than the unit disk graphs (UDGs) and approximate disk graphs (ADGs) studied in Chapter 8. Here again, an alternate approach to

our MIS-tree-based construction has to be searched for, since it highly relies on geometrical properties of UDGs and ADGs. Finally, after proving their correctness, analyzing their complexity, and simulating their functioning, we would like to go a step further in the study of our algorithms by deploying them on real wireless sensor networks. This would allow to face possible implementation issues, to measure their efficiency, and to study their impact on energy consumption.

(f, g)-Alliance. Our self-stabilizing solution with safe convergence to the problem of constructing a minimal (f, g)-alliance in the case $f \geq g$ in Chapter 10 raises several other questions. Since the notion of (f, g)-alliance generalizes some other spanning structures, we expect that there are many implications of the actual results on well-known instances to the study of (f, g)-alliance construction, and conversely. For example, impossibility results and complexity bounds on the construction of dominating set would also apply to the one of (f, g)-alliance. This would also help us to improve the time complexity of our solution, without compromising its space complexity. Finally, there are two other cases we did not investigate in our work. Is it possible to build an (f, g)-alliance in the case $f < g$ efficiently? The same question comes about the case where f and g do not satisfy the same inequality for every process of the network.

Other Skylines. More generally speaking, it would be interesting to study other properties derived from self-stabilization. Particularly, self-stabilizing construction of spanning structures with fault-containment is very attractive, since it aims at confining the faults in a small part of the network. Ideally, this small part would match a subdivision of the spanning structures being built. Note that all our work is written in the locally shared memory model. A direct extension of our work would be to propose *efficient* self-stabilizing solutions for the same problems in the message-passing model. This model has the advantage of being close to implementation, so the efficiency of the proposed solutions should be the main goal. Note that general but costly constructions already exist. Ultimately, it would be interesting to go beyond the study of undirected network topologies, that is, to consider the same spanning structures in directed network topologies when it makes sense.

Bibliography

[AAER07] Dana Angluin, James Aspnes, David Eisenstat, and Eric Ruppert. The computational power of population protocols. *Distributed Computing*, 20(4):279–304, 2007. (Cited on pages xxviii and 114.)

[AB93] Yehuda Afek and Geoffrey M. Brown. Self-stabilization over unreliable communication media. *Distributed Computing*, 7(1):27–34, 1993. (Cited on page 18.)

[ALRL04] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl E. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004. (Cited on pages xiii and 13.)

[APHV00] Alan D. Amis, Ravi Prakash, Dung Huynh, and Thai Vuong. Max-Min *D*-Cluster Formation in Wireless Ad Hoc Networks. In *IEEE International Conference on Computer Communications (INFOCOM)*, pages 32–41. IEEE Computer Society Press, 2000. (Cited on pages xxiii and 73.)

[AV91] Baruch Awerbuch and George Varghese. Distributed program checking: a paradigm for building self-stabilizing distributed protocols. In *Proceedings of the 32nd Annual Symposium on Foundations of Computer Science*, pages 258–267, San Juan, 1991. IEEE Computer Society Press. (Cited on page 17.)

[AW91] Hagit Attiya and Jennifer L. Welch. Sequential consistency versus linearizability (extended abstract). In *Proceedings of the 3rd Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA, pages 304–315, Hilton Head, South Carolina, USA, July 21–24 1991. ACM. (Cited on page 18.)

[Bar64] Paul Baran. On distributed communications networks. *IEEE Transactions on Communications Systems*, 12(1):1–9, 1964. (Cited on pages ix and 9.)

[Bas99] Stefano Basagni. Distributed clustering for ad hoc networks. In *International Symposium on Parallel Architectures, Algorithms and Networks*, ISPAN '99, pages 310–315, Fremantle, Australia, June 23–25 1999. (Cited on pages xii and 13.)

[BDN95] Brian Bourgon, Ajoy K. Datta, and Viruthagiri Natarajan. A self-stabilizing ranking algorithm for tree structured networks. In *Proceedings of the First Workshop on Self-Stabilizing Systems (WSS'95)*, pages 23–28, 1995. (Cited on pages xxxii and 141.)

[BDV05] Doina Bein, Ajoy K. Datta, and Vincent Villain. Snap-stabilizing optimal binary-search-tree. In *Proceedings of the 7-th International Symposium on Self-Stabilizing Systems (SSS)*, Lecture Notes in Computer Science (LNCS), pages 1–17, Barcelona, Spain, October 26–27 2005. Springer. (Cited on pages xxxii and 141.)

[BFN01] Lali Barrière, Pierre Fraigniaud, and Lata Narayanan. Robust position-based routing in wireless ad hoc networks with unstable transmission ranges. In *Proceedings of the 5th International Workshop on Discrete Algorithms and Methods for Mobile Computing and Communications (DIAL-M)*, pages 19–27. ACM, 2001. (Cited on page 22.)

[BGJ01] Joffroy Beauquier, Maria Gradinariu, and Colette Johnen. Cross-over composition - enforcement of fairness under unfair adversary. In *Workshop on Self-Stabilizing Systems (WSS)*, pages 19–34, 2001. (Cited on pages xx and 37.)

[BHKPS05] Janna Burman, Ted Herman, Shay Kutten, and Boaz Patt-Shamir. Asynchronous and fully self-stabilizing time-adaptive majority consensus. In *Revised Selected Papers of the 9th International Conference on Principles of Distributed Systems (OPODIS 2005), Pisa, Italy, December 12–14, 2005*, volume 3974 of *Lecture Notes in Computer Science (LNCS)*, pages 146–160. Springer Berlin Heidelberg, 2005. (Cited on page 16.)

[BK01] Suman Banerjee and Samir Khuller. A clustering scheme for hierarchical control in multi-hop wireless networks. In *IEEE International Conference on Computer Communications (INFOCOM)*, pages 1028–1037, 2001. (Cited on pages xii and 13.)

[BND89] Amotz Bar-Noy and Danny Dolev. Shared-memory vs. message-passing in an asynchronous distributed environment. In *Proceedings of the eighth annual ACM Symposium on Principles of distributed computing*, PODC '89, pages 307–318, Edmonton, Alberta, Canada, 1989. ACM. (Cited on page 18.)

[Bou07] Christian Boulinier. *L'Unisson*. PhD thesis, Université de Picardie Jules Verne, Amiens, October 2007. Dissertation in french. (Cited on page 39.)

[BPV04] Christian Boulinier, Franck Petit, and Vincent Villain. When graph theory helps self-stabilization. In *ACM Symposium on Principles of Dis-*

tributed Computing (PODC), pages 150–159, 2004. (Cited on pages xx, 38 and 39.)

[Bru78] Peter Brucker. On the complexity of clustering problems. In *Optimization and Operations Research*, volume 157 of *Lecture Notes in Economics and Mathematical Systems*, pages 45–54. Springer Berlin Heidelberg, 1978. (Cited on pages xxiii and 70.)

[Bur02] Arthur W. Burks. The invention of the universal electronic computer–how the electronic computer revolution began. *Future Generation Comp. Syst.*, 18(7):871–892, 2002. (Cited on pages ix and 9.)

[CD94] Zeev Collin and Shlomi Dolev. Self-stabilizing depth-first search. *Information Processing Letters*, 49:297–301, 1994. (Cited on pages xxi and 106.)

[CDD+13] Fabienne Carrier, Ajoy K. Datta, Stéphane Devismes, Lawrence L. Larmore, and **Yvan Rivierre**. Self-Stabilizing (f, g)-Alliances with Safe Convergence. In *Proceedings of the 15th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, volume 8255 of *Lecture Notes in Computer Science (LNCS)*, pages 61–75, Osaka, Japan, November 13–16 2013. Springer. (Cited on page 4.)

[CDDL10] Eddy Caron, Ajoy K. Datta, Benjamin Depardon, and Lawrence L. Larmore. A Self-Stabilizing k-Clustering Algorithm for Weighted Graphs. *Journal of Parallel and Distributed Computing (JPDC)*, 70(11):1159–1173, 2010. (Cited on pages xxiii and 73.)

[CL85] Kanianthra Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems (TOCS)*, 3(1):63–75, 1985. (Cited on page 17.)

[CM84] K. Mani Chandy and Jayadev Misra. The drinking philosopher's problem. *Transactions on Programming Languages and Systems (TOPLAS)*, 6(4):632–646, 1984. (Cited on pages xi and 11.)

[Coo79] Stephen A. Cook. Deterministic CFL's are Accepted Simultaneously in Polynomial Time and Log Squared Space. In *STOC*, pages 338–345. ACM, 1979. (Cited on page 51.)

[Coo85] Stephen A. Cook. A Taxonomy of Problems with Fast Parallel Algorithms. *Information and Control*, 64:2–22, March 1985. International Conference on Foundations of Computation Theory (FCT). (Cited on page 51.)

[Cri63] Arthur J. Critchlow. Generalized multiprocessing and multiprogramming systems. In *Proceedings of the November 12–14, 1963, fall joint computer*

conference, AFIPS '63 (Fall), pages 107–126, Las Vegas, Nevada, 1963. ACM. (Cited on pages ix and 9.)

[CT05a] Pranay Chaudhuri and Hussein Thompson. A self-stabilizing algorithm for l(2, 1)-labeling trees. In *Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Networks, part of the 23rd Multi-Conference on Applied Informatics*, Parallel and Distributed Computing and Networks, pages 627–632, Innsbruck, Austria, February 15–17 2005. IASTED/ACTA Press. (Cited on page 57.)

[CT05b] Pranay Chaudhuri and Hussein Thompson. Self-stabilizing tree ranking. *International Journal of Computer Mathematics*, 82(5):529–539, 2005. (Cited on pages xxii and 57.)

[CYH91] Nian-Shing Chen, Hwey-Pyng Yu, and Shing-Tsaan Huang. A self-stabilizing algorithm for constructing spanning trees. *Information Processing Letters*, 39:147–151, 1991. (Cited on pages xxi and 106.)

[DDH+11a] Ajoy K. Datta, Stéphane Devismes, Karel Heurtefeux, Lawrence L. Larmore, and **Yvan Rivierre**. Algorithme auto-stabilisant construisant un ensemble k-dominant minimal borné. In *Rencontres francophones du Parallélisme (RenPar'20)*, Saint-Malo, France, May 10–13 2011. (Cited on page xxvii.)

[DDH+11b] Ajoy K. Datta, Stéphane Devismes, Karel Heurtefeux, Lawrence L. Larmore, and **Yvan Rivierre**. Self-Stabilizing Small k-Dominating Sets. In *The Second International Conference on Networking and Computing (ICNC)*, pages 30–39. IEEE Computer Society Press, 2011. Best Paper Award. (Cited on page 3.)

[DDH+12] Ajoy K. Datta, Stéphane Devismes, Karel Heurtefeux, Lawrence L. Larmore, and **Yvan Rivierre**. Competitive Self-Stabilizing k-Clustering. In *2012 IEEE 32nd International Conference on Distributed Computing Systems (ICDCS)*, pages 476–485, Macau, China, June 18–21 2012. IEEE Computer Society Press. (Cited on page 3.)

[DDH+13] Ajoy K. Datta, Stéphane Devismes, Karel Heurtefeux, Lawrence L. Larmore, and **Yvan Rivierre**. Self-Stabilizing Small k-Dominating Sets. *International Journal of Networking and Computing (IJNC)*, 3(1):116–136, 2013. (Cited on page 3.)

[DDL09] Ajoy K. Datta, Stéphane Devismes, and Lawrence L. Larmore. A Self-Stabilizing $O(n)$-Round k-Clustering Algorithm. In *International Symposium on Reliable Distributed Systems (SRDS)*, pages 147–155. IEEE Computer Society Press, 2009. (Cited on pages xxiii, xxiv, xxvii, 73, 75, 84, 85, 105, 107 and 109.)

[DDLR11] AjoyK. Datta, Stéphane Devismes, Lawrence L. Larmore, and **Yvan Rivierre**. Self-stabilizing labeling and ranking in ordered trees. In *Proceedings of the 13th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, volume 6976 of *Lecture Notes in Computer Science (LNCS)*, pages 148–162. Springer Berlin Heidelberg, Grenoble, France, October 10–12 2011. (Cited on page 4.)

[DDLR13] AjoyK. Datta, Stéphane Devismes, Lawrence L. Larmore, and **Yvan Rivierre**. Self-stabilizing labeling and ranking in ordered trees. *Theoretical Computer Science (TCS)*, 2013. To appear. (Cited on page 4.)

[DDT05] Sylvie Delaët, Bertrand Ducourthial, and Sébastien Tixeuil. Self-stabilization with r-operators revisited. In *Self-Stabilizing Systems*, pages 68–80, 2005. (Cited on page 18.)

[DGPV01] Ajoy K. Datta, Shivashankar Gurumurthy, Franck Petit, and Vincent Villain. Self-stabilizing network orientation algorithms in arbitrary rooted networks. *Studia Informatica Universalis*, 1(1):1–22, 2001. (Cited on page 57.)

[DGS96] Shlomi Dolev, Mohamed G. Gouda, and Marco Schneider. Memory Requirements for Silent Stabilization. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 27–34, 1996. (Cited on page 33.)

[DH97] Shlomi Dolev and Ted Herman. Superstabilizing protocols for dynamic distributed systems. *Chicago Journal of Theoretical Computer Science*, 1997, 1997. (Cited on page 16.)

[Dij65] Edsger W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, 1965. (Cited on pages xi and 11.)

[Dij74] Edsger W. Dijkstra. Self-Stabilizing Systems in Spite of Distributed Control. *Communications of the ACM*, 17:643–644, 1974. (Cited on pages vii, xv, xvii, 2, 15 and 26.)

[DIM89] Shlomi Dolev, Amos Israeli, and Shlomo Moran. Self stabilization of dynamic systems. In *The Sixteenth Conference of Electrical and Electronics Engineers in Israel*, 1989. (Cited on pages xv and 15.)

[DIM93] Shlomi Dolev, Amos Israeli, and Shlomo Moran. Self-stabilization of dynamic systems assuming only read/write atomicity. *Distributed Computing*, 7(1):3–16, 1993. (Cited on pages xvii and 26.)

[DLV09] Ajoy K. Datta, Lawrence L. Larmore, and Priyanka Vemula. A Self-Stabilizing $O(k)$-Time k-Clustering Algorithm. *The Computer Journal*, page bxn071, 2009. (Cited on pages xxiii and 73.)

[DLV11a] Ajoy K. Datta, Lawrence L. Larmore, and Priyanka Vemula. An o(n)-time self-stabilizing leader election algorithm. *Journal of Parallel and Distributed Computing (JPDC)*, 71(11):1532–1544, 2011. (Cited on pages xxi, xxiv, 45 and 49.)

[DLV11b] Ajoy K. Datta, Lawrence L. Larmore, and Priyanka Vemula. Self-stabilizing leader election in optimal space under an arbitrary scheduler. *Theoretical Computer Science*, 412(40):5541–5561, 2011. (Cited on pages 78 and 85.)

[Dol00] Shlomi Dolev. *Self-Stabilization*. MIT Press, 2000. (Cited on pages xx and 35.)

[DP12] Stéphane Devismes and Franck Petit. On efficiency of unison. In *Proceedings of the 4th International Workshop on Theoretical Aspects of Dynamic Distributed Systems*, TADDS '12, pages 20–25, Roma, Italy, 2012. ACM. (Cited on page 39.)

[DPRS11] Mitre Costa Dourado, Lucia Draque Penso, Dieter Rautenbach, and Jayme Luiz Szwarcfiter. The south zone: Distributed algorithms for alliances. In *International Symposium on Self-Stabilizing Systems (SSS)*, pages 178–192, 2011. (Cited on pages xxviii, xxix, 114 and 115.)

[DS97] Hochbaum Dorit S. *Approximation algorithms for NP-hard problems*. PWS Publishing Co., Boston, MA, USA, 1997. (Cited on page 72.)

[FEP+06] Paola Flocchini, Antonio Mesa Enriques, Linda Pagli, Giuseppe Prencipe, and Nicola Santoro. Point-of-failure shortest-path rerouting: Computing the optimal swap edges distributively. *IEICE Transactions on Information and Systems*, 89-D(2):700–708, 2006. (Cited on pages xxii, 57 and 58.)

[FG69] Jon H. Folkman and Ronald L. Graham. An Inequality in the Geometry of Numbers. *Canadian Mathematical Bulletin (CMB)*, 12:745–752, 1969. (Cited on pages xxvi and 102.)

[FLM85] Michael J. Fischer, Nancy A. Lynch, and Michael Merritt. Easy impossibility proofs for distributed consensus problems. In *Proceedings of the Fourth Annual ACM Symposium on Principles of Distributed Computing*, PODC '85, pages 59–70, Minaki, Ontario, Canada, August 5–7 1985. (Cited on pages xi and 12.)

[FLP85] Michael J. Fischer, Nancy A. Lynch, and Mike Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985. (Cited on pages xi, xiv, 12 and 15.)

[Flu08] Roland Flury. Sinalgo – simulator for network algorithms. Distributed Computing Group (DCG) – Eidgenössische Technische Hochschule

(ETH) Zurich, 2008. http://dcg.ethz.ch/projects/sinalgo. (Cited on pages xxvii and 105.)

[FM02] Yaacov Fernandess and Dahlia Malkhi. K-Clustering in Wireless Ad Hoc Networks. In *ACM Workshop on Principles of Mobile Computing (POMC)*, pages 31–37, 2002. (Cited on pages xii, xxi, xxiii, 13, 43, 72 and 73.)

[FR07] Henning Fernau and Daniel Raible. Alliances in graphs: a complexity-theoretic study. In *Proceedings of the 33rd Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM 2007), Part II*, Theory and Practice of Computer Science, pages 61–70, Harrachov, Czech Republic, January 20–26 2007. Institute of Computer Science AS CR, Prague. (Cited on pages xiii and 13.)

[Gär03] Felix C. Gärtner. A survey of self-stabilizing spanning-tree construction algorithms. Technical Report 38, Swiss Federal Institute of Technology (EPFL), 2003. (Cited on pages xii and 13.)

[GGHP96] Sukumar Ghosh, Arobinda Gupta, Ted Herman, and Sriram V. Pemmaraju. Fault-containing self-stabilizing algorithms. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing, Philadelphia, Pennsylvania, USA, May 23–26, 1996*, pages 45–54. ACM, 1996. (Cited on page 16.)

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979. (Cited on pages xxii and 71.)

[GM91] Mohamed G. Gouda and Nicolas J. Multari. Stabilizing communication protocols. *IEEE Transactions on Computers (TC)*, 40(4):448–458, 1991. (Cited on page 18.)

[GMOR05] Anupam Gupta, Bruce M. Maggs, Florian Oprea, and Michael K. Reiter. Quorum placement in networks to minimize access delays. In *Proceedings of the Twenty-Fourth Annual ACM Symposium on Principles of Distributed Computing*, PODC 2005, pages 87–96, Las Vegas, NV, USA, July 17–20 2005. ACM. (Cited on pages xxviii and 114.)

[GS84] David K. Gifford and Alfred Z. Spector. The space shuttle primary computer system. *Communications of the ACM*, 27(9):872–900, 1984. (Cited on pages xiv and 15.)

[GT02] Christophe Genolini and Sébastien Tixeuil. A lower bound on dynamic k-stabilization in asynchronous systems. In *Proceedings of the 21st Symposium on Reliable Distributed Systems*, SRDS, pages 212–221, Osaka, Japan, October 13–16 2002. (Cited on pages xvi and 17.)

[HC92] Shing-Tsaan Huang and Nian-Shing Chen. A Self-Stabilizing Algorithm for Constructing Breadth-First Trees. *Information Processing Letters*, 41:109–117, 1992. (Cited on pages xxi, xxvii, 45 and 106.)

[Her92] Ted Herman. *Adaptivity through distributed convergence*. PhD thesis, University of Texas at Austin, Austin, TX, USA, 1992. UMI Order No. GAX92-12547. (Cited on pages xx, 35 and 36.)

[HL90] Stephen T. Hedetniemi and Renu C. Laskar. Bibliography on domination in graphs and some basic definitions of domination parameters. *Discrete Mathematics*, 86(1-3):257–277, 1990. (Cited on pages xiii and 13.)

[HLCW07] Tetz C. Huang, Ji-Cherng Lin, Chih-Yuan Chen, and Cheng-Pin Wang. A self-stabilizing algorithm for finding a minimal 2-dominating set assuming the distributed demon model. *Computers & Mathematics with Applications*, 54(3):350–356, 2007. (Cited on pages xxiii and 70.)

[HM01] Ted Herman and Toshimitsu Masuzawa. A stabilizing search tree with availability properties. In *Fifth International Symposium on Autonomous Decentralized Systems*, ISADS 2001, pages 398–405, 2001. (Cited on pages xxxii and 141.)

[HNM02] Rodney R. Howell, Mikhail Nesterenko, and Masaaki Mizuno. Finite-state self-stabilizing protocols in message-passing systems. *Journal of Parallel and Distributed Computing (JPDC)*, 62(5):792–817, 2002. (Cited on page 18.)

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985. (Cited on pages xi and 11.)

[HP86] James Orchard Halliwell-Phillipps. The story of the three little pigs. In *The Nursery Rhymes of England*, pages 37–41. London and New York: Frederick Warne and Company, 1886. (Cited on page 1.)

[HP01] Ted Herman and Imran A. Pirwani. A composite stabilizing data structure. *5th International Workshop on Self-Stabilizing Systems (WSS 2001), Lecture Notes in Computer Science LNCS 2194*, pages 167–182, 2001. (Cited on pages xxxii and 141.)

[IKK02] Michiyo Ikeda, Sayaka Kamei, and Hirotsugu Kakugawa. A space-optimal self-stabilizing algorithm for the maximal independent set problem. In *International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT)*, pages 70–74, 2002. (Cited on pages xxiii and 73.)

[Kat93] Shmuel Katz. A superimposition control construct for distributed systems. *Transactions on Programming Languages and Systems (TOPLAS)*, 15(2):337–356, 1993. (Cited on page 17.)

[KHH04] Petter Kristiansen, Sandra M. Hedetniemi, and Stephen T. Hedetniemi. Alliances in graphs. *Journal of Combinatorial Mathematics and Combinatorial Computing (JCMCC)*, 48:157–178, 2004. (Cited on pages xiii and 13.)

[KK03] Sayaka Kamei and Hirotsugu Kakugawa. A self-stabilizing algorithm for the distributed minimal k-redundant dominating set problem in tree networks. In *Proceedings of the Fourth International Conference on Parallel and Distributed Computing, Applications and Technologies*, PDCAT'2003, pages 720–724, 2003. (Cited on pages xxiii, 70 and 177.)

[KK06] Adrian Kosowski and Lukasz Kuszner. Energy optimisation in resilient self-stabilizing processes. In *Proceedings of the Fifth International Conference on Parallel Computing in Electrical Engineering*, PARELEC, pages 105–110, Bialystok, Poland, September 13–17 2006. (Cited on pages xx and 37.)

[KK07] Sayaka Kamei and Hirotsugu Kakugawa. A self-stabilizing approximation algorithm for the minimum weakly connected dominating set with safe convergence. In *Proceedings of the First International Workshop on Reliability, Availability, and Security*, WRAS, pages 57–67, Paris, France, September 2007. (Cited on pages xvi, xix and 17.)

[KK12] Sayaka Kamei and Hirotsugu Kakugawa. A self-stabilizing 6-approximation for the minimum connected dominating set with safe convergence in unit disk graphs. *Theoretical Computer Science (TCS)*, 428:80–90, 2012. (Cited on pages xvi, xix and 17.)

[KM06] Hirotsugu Kakugawa and Toshimitsu Masuzawa. A self-stabilizing minimal dominating set algorithm with safe convergence. In *International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE Computer Society Press, 2006. (Cited on pages xvi, xix, xxix, 17 and 115.)

[KP95a] Shay Kutten and David Peleg. Fast Distributed Construction of k-Dominating Sets and Applications. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 238–251, 1995. (Cited on pages xxiv, 73 and 76.)

[KP95b] Shay Kutten and David Peleg. Fault-local distributed mending (extended abstract). In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '95, pages 20–27, Ottowa, Ontario, Canada, 1995. ACM. (Cited on page 16.)

[KPS97] Shay Kutten and Boaz Patt-Shamir. Time-adaptive self stabilization. In *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '97, pages 149–158, Santa Barbara, California, USA, August 21–24 1997. ACM. (Cited on page 16.)

[KWZ03] Fabian Kuhn, Roger Wattenhofer, and Aaron Zollinger. Ad-hoc networks beyond unit disk graphs. In *DIALM-POMC*, pages 69–78. ACM, 2003. (Cited on page 22.)

[Lad75] Richard E. Ladner. The Circuit Value Problem is Log Space Complete for *P*. *SIGACT News*, 7:18–20, January 1975. (Cited on page 52.)

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978. (Cited on pages xi and 11.)

[Lam85] Leslie Lamport. Solved problems, unsolved problems and non-problems in concurrency. *SIGOPS Operating Systems Review*, 19(4):34–44, 1985. (Cited on pages xv and 15.)

[LL77] Gérard Le Lann. Distributed systems-towards a formal approach. In *Information Processing (IFIP) Congress*, volume 7, pages 155–160. North-Holland Publishing Company, 1977. (Cited on pages xi and 12.)

[LSP82] Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. The byzantine generals problem. *Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982. (Cited on pages xiv and 14.)

[OR00] Rafail Ostrovsky and Yuval Rabani. Polynomial time approximation schemes for geometric k-clustering. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, pages 349–358, 2000. (Cited on pages xxiii and 70.)

[PB04] Lucia Draque Penso and Valmir C. Barbosa. A Distributed Algorithm to Find k-Dominating Sets. *Discrete Applied Mathematics*, 141(1–3):243–253, 2004. (Cited on pages xxiv, 73 and 76.)

[PU89] David Peleg and Eli Upfal. A Trade-Off between Space and Efficiency for Routing Tables. *Journal of the ACM*, 36(3):510–530, 1989. (Cited on pages xxiii, xxiv, 73, 75 and 76.)

[Raj02] Rajmohan Rajaraman. Topology control and routing in ad hoc networks: a survey. *SIGACT News*, 33(2):60–73, 2002. (Cited on pages x and 11.)

[Rav05] Vlady Ravelomanana. Distributed k-Clustering Algorithms for Random Wireless Multihop Networks. In *Proceedings of the 4th International Conference on Networking (ICN), Part I*, pages 109–116, 2005. (Cited on pages xxiii, xxiv and 73.)

[SGLA04] Marco Aurélio Spohn and Jose Joaquin Garcia-Luna-Aceves. Bounded-Distance Multi-Clusterhead Formation in Wireless Ad Hoc Networks. *Ad Hoc Networks*, 5:504–530, 2004. (Cited on pages xxiii and 73.)

[SMP99] Katayoun Sohrabi, Bertha Manriquez, and Gregory J Pottie. Near ground wideband channel measurement in 800-1000 mhz. In *Proceedings of the 49th Vehicular Technology Conference*, volume 1 of *VTC*, pages 571–574. IEEE, 1999. (Cited on page 22.)

[SRR95] Sandeep K. Shukla, Daniel J. Rosenkrantz, and S. Sekharipuram Ravi. Observations on Self-Stabilizing Graph Algorithms on Anonymous Networks. In *Workshop on Self-Stabilizing Systems (WSS)*, pages 7.1–7.15, 1995. (Cited on pages xxiii and 73.)

[SX07] Pradip K. Srimani and Zhenyu Xu. Distributed protocols for defensive and offensive alliances in network graphs using self-stabilization. In *International Conference on Computer Theory and Applications (ICCTA)*, pages 27–31. IEEE Computer Society Press, 2007. (Cited on pages xxix and 115.)

[Tel01] Gerard Tel. *Introduction to Distributed Algorithms*. Cambridge University Press, 2nd edition, 2001. (Cited on pages xix and 35.)

[Tur07] Volker Turau. Linear self-stabilizing algorithms for the independent and dominating set problems using an unfair distributed scheduler. *Information Processing Letters*, 103(3):88–93, 2007. (Cited on pages xxix and 115.)

[Var94] George Varghese. Self-stabilization by counter flushing. In *Proceedings of the thirdteenth annual ACM Symposium on Principles of Distributed Computing*, PODC '94, pages 244–253, 1994. (Cited on page 18.)

[Wil64] J. W. J. Williams. Algorithm 232: Heapsort. *Communications of the ACM*, 7(6):347–348, 1964. (Cited on page 142.)

[WWTZ12] Guangyuan Wang, Hua Wang, Xiaohui Tao, and Ji Zhang. A self-stabilizing algorithm for finding a minimal k-dominating set in general networks. In *Proceedings of the Third International Conference on Data and Knowledge Engineering*, volume 7696 of *Lecture Notes in Computer Science (LNCS)*, pages 74–85, Wuyishan, Fujian, China, November 21–23 2012. Springer. In this paper, the term "k-dominating set" actually refers to a "k-redundant dominating set" as introduced in [KK03]. (Cited on pages xxiii, xxix, 70 and 115.)

Index

Résumé

Cette thèse s'intéresse à la construction auto-stabilisante de structures couvrantes dans un système réparti. L'auto-stabilisation est un paradigme pour la tolérance aux fautes dans les algorithmes répartis. Plus précisément, elle garantit que le système retrouve un comportement correct en temps fini après avoir été perturbé par des fautes transitoires.

Notre modèle de système réparti se base sur des mémoires localement partagées pour la communication, des identifiants uniques pour briser les symétries et un ordonnanceur inéquitable, c'est-à-dire le plus faible des ordonnanceurs. Dans la mesure du possible, nous nous imposons d'utiliser les plus faibles hypothèses, afin d'obtenir les constructions les plus générales de structures couvrantes réparties.

Nous présentons quatre algorithmes auto-stabilisants originaux pour le k-partitionnement, la construction d'une (f, g)-alliance et l'indexation. Pour chacun de ces problèmes, nous prouvons la correction de nos solutions. De plus, nous analysons leur complexité en temps et en espace à l'aide de preuves formelles et de simulations. Enfin, pour le problème de (f, g)-alliance, nous prenons en compte la notion de convergence sûre qui vient s'ajouter à celle d'auto-stabilisation. Elle garantit d'abord que le comportement du système assure rapidement un minimum de conditions, puis qu'il continue de converger jusqu'à se conformer à une spécification plus exigeante.

Mots-clés : Auto-stabilisation, convergence sûre, algorithme réparti, k-partitionnement, ensemble k-dominant, (f, g)-alliance.

Abstract

This thesis deals with the self-stabilizing construction of spanning structures over a distributed system. Self-stabilization is a paradigm for fault-tolerance in distributed algorithms. It guarantees that the system eventually satisfies its specification after transient faults hit the system.

Our model of distributed system assumes locally shared memories for communicating, unique identifiers for symmetry-breaking, and distributed daemon for execution scheduling, that is, the weakest proper daemon. More generally, we aim at the weakest possible assumptions, such as arbitrary topologies, in order to propose the most versatile constructions of distributed spanning structures.

We present four original self-stabilizing algorithms achieving k-clustering, (f, g)-alliance construction, and ranking. For each of these problems, we prove the correctness of our solutions. Moreover, we analyze their time and space complexity using formal proofs and simulations. Finally, for the (f, g)-alliance problem, we consider the notion of safe convergence in addition to self-stabilization. It enforces the system first to satisfy a specification that guarantees a minimum of conditions quickly, and then to converge to a more stringent specification.

Keywords: Self-stabilization, safe convergence, distributed algorithm, k-clustering, k-dominating set, (f, g)-alliance.